Software Security

CSIE 7190 Cryptography and Network Security, Spring 2018

https://ceiba.ntu.edu.tw/1062csie_cns

cns@csie.ntu.edu.tw

Hsu-Chun Hsiao

Agenda

Background

Memory-safety vulnerabilities

- Buffer overflow vulnerabilities
- Integer conversion vulnerabilities
- Format string vulnerabilities

Automated program analysis

- Fuzzing
- Symbolic execution

Software Vulnerabilities

Software vulnerabilities could cause severe damage.

For example:

- Use injection attack to dump sensitive data
- Exploit buffer overflow vulnerability to perform unauthorized remote login.
- Bypass certificate verification through the inconsistency of TLS protocol and corresponding implementation.

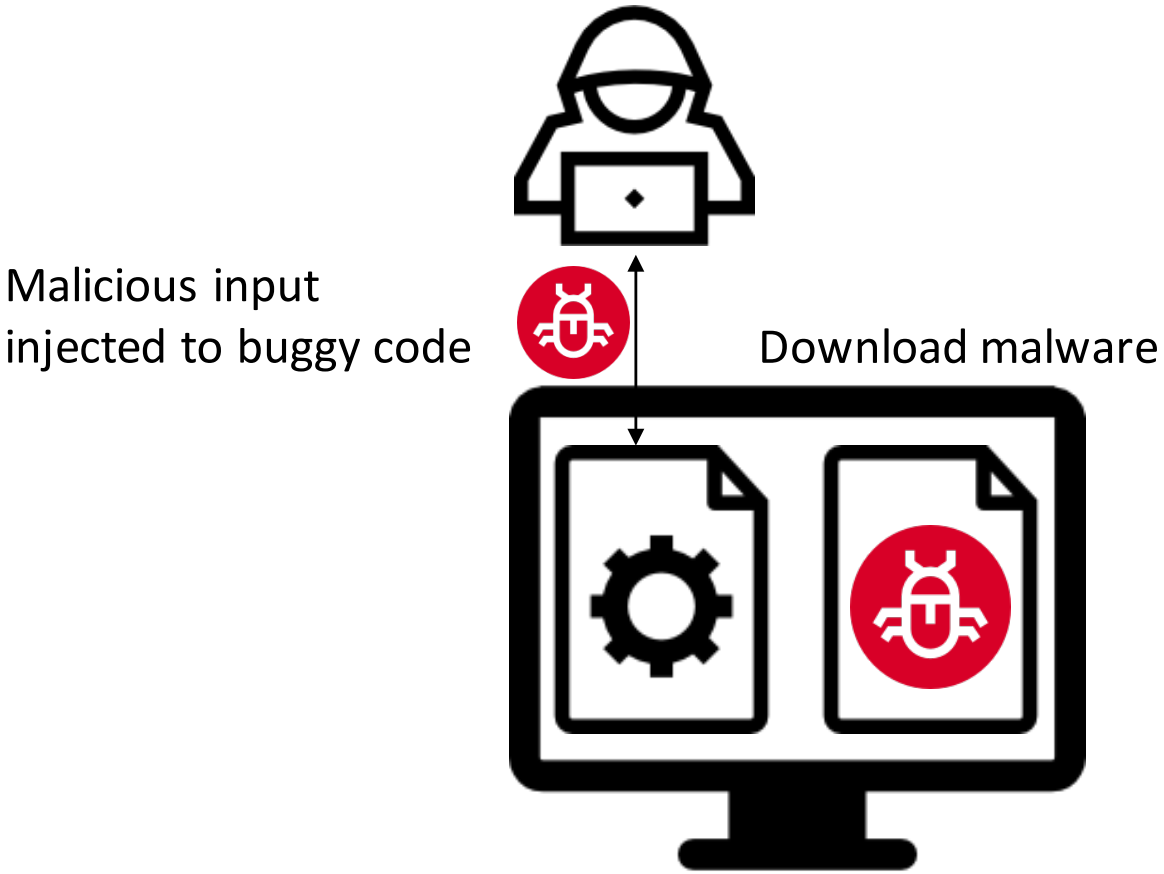
Focus of software security

Vulnerabilities in software implementation

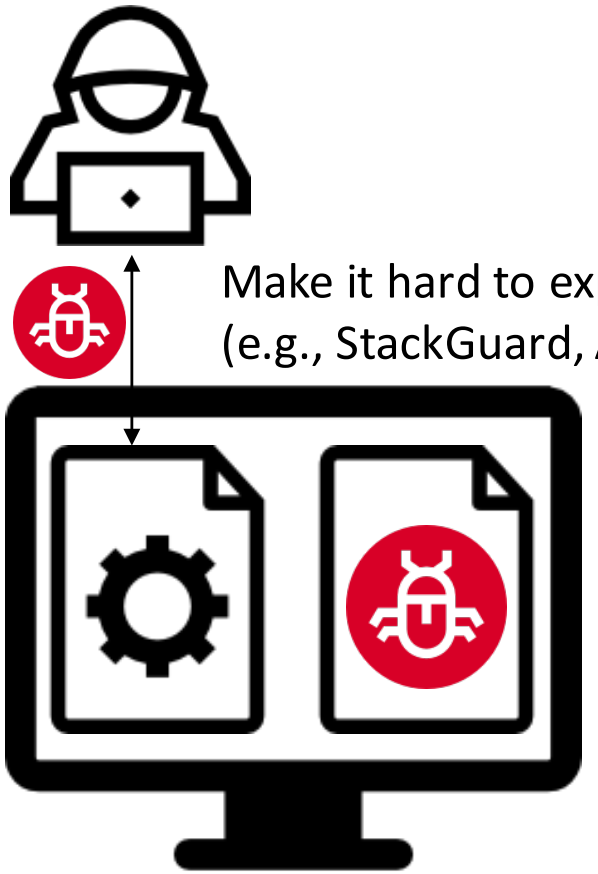
Attacks that exploit these vulnerabilities

Defenses against these attacks

Common Attack Vectors



Common Defense Tactics



Make it hard to exploit
(e.g., StackGuard, ASLR)

Eliminate bugs
(e.g., type-safe language, formal methods, software testing)

Prevent malware injection/ download
(e.g., firewall, WAF)

Detect execution of malicious code
(e.g., antivirus, behavior monitoring)

Ensure running only trusted code
(e.g., attestation, trusted computing)

Memory-safety vulnerabilities

Unauthorized read or write to memory locations

Most common type of software vulnerabilities

Attacker's goals

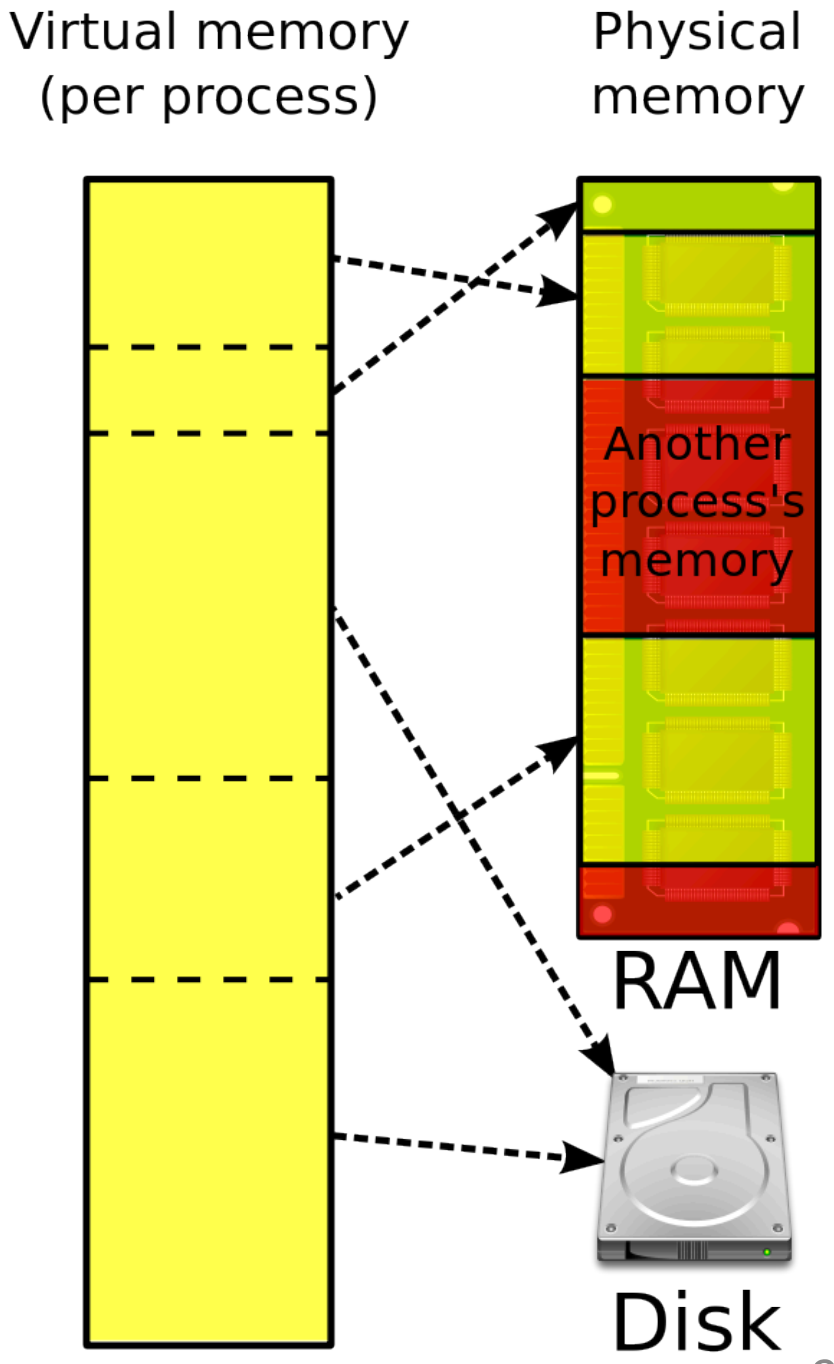
- Steal sensitive info
- Inject malicious code
- Execute malicious code
- Take control over the machine

Many examples in this lecture borrowed from Prof. Dawn Song's notes

Background: Memory Layout

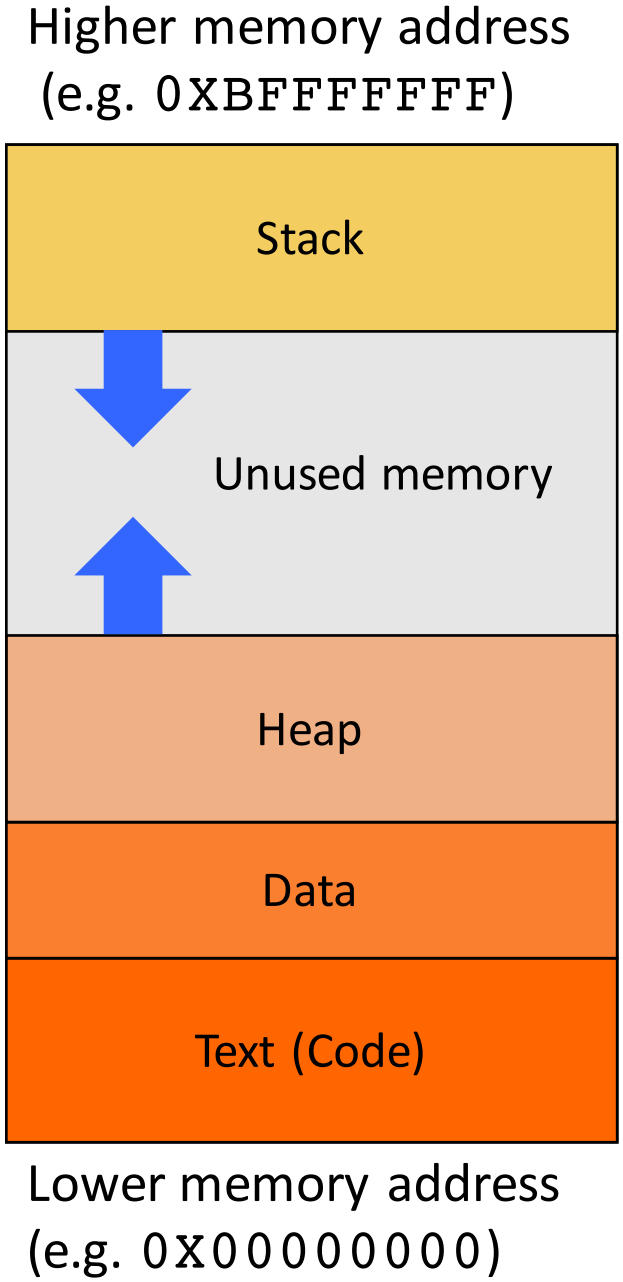
Note: details are machine- and OS- dependent

Per-process virtual memory



http://en.wikipedia.org/wiki/Virtual_memory

Per-process virtual memory layout



Stack

- Stack frames of function calls
- Local variables, parameter, return address

Heap

- Dynamic allocated data

Data segment

- Global and static variables
- Initialized or uninitialized

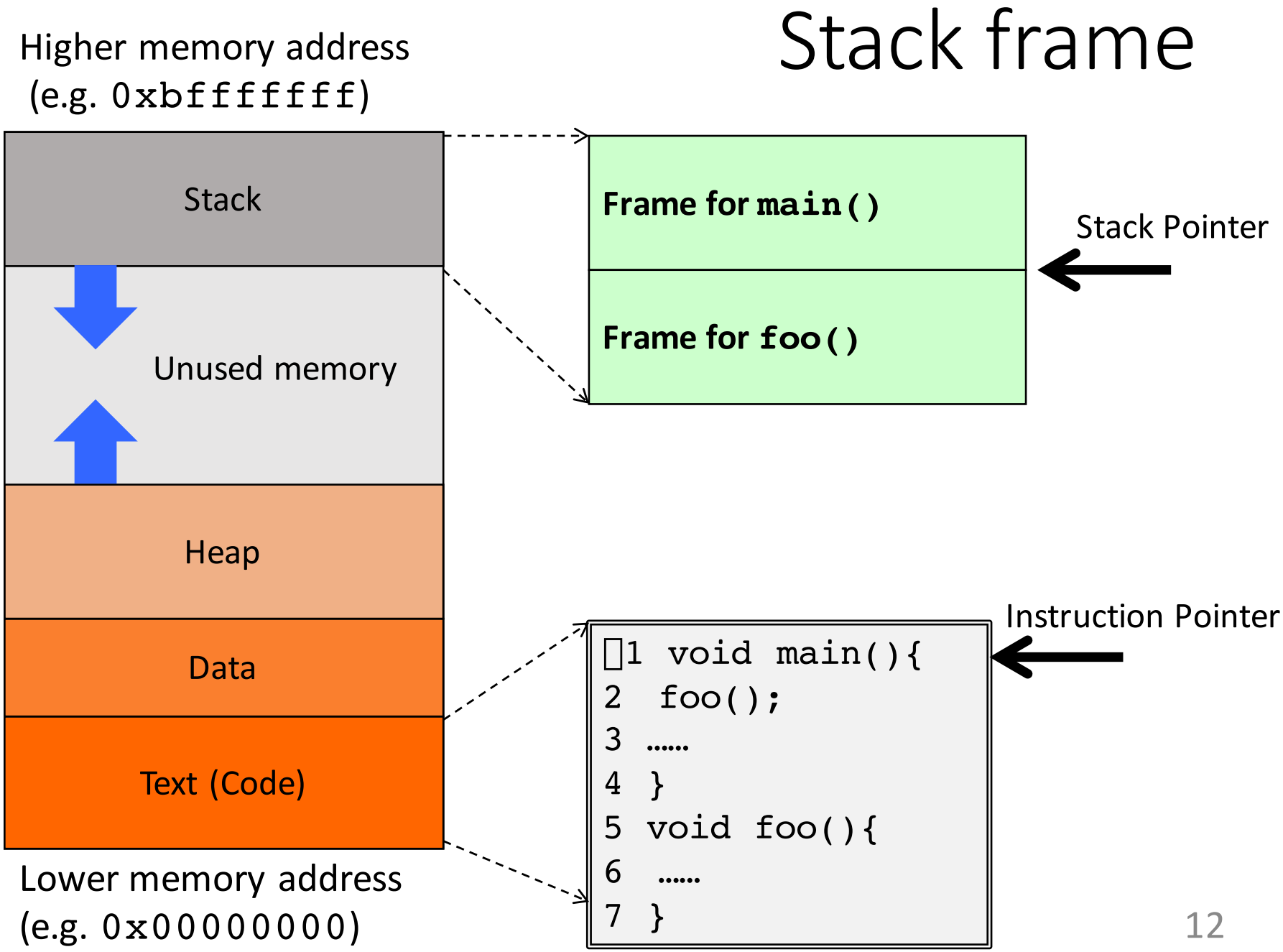
Text segment

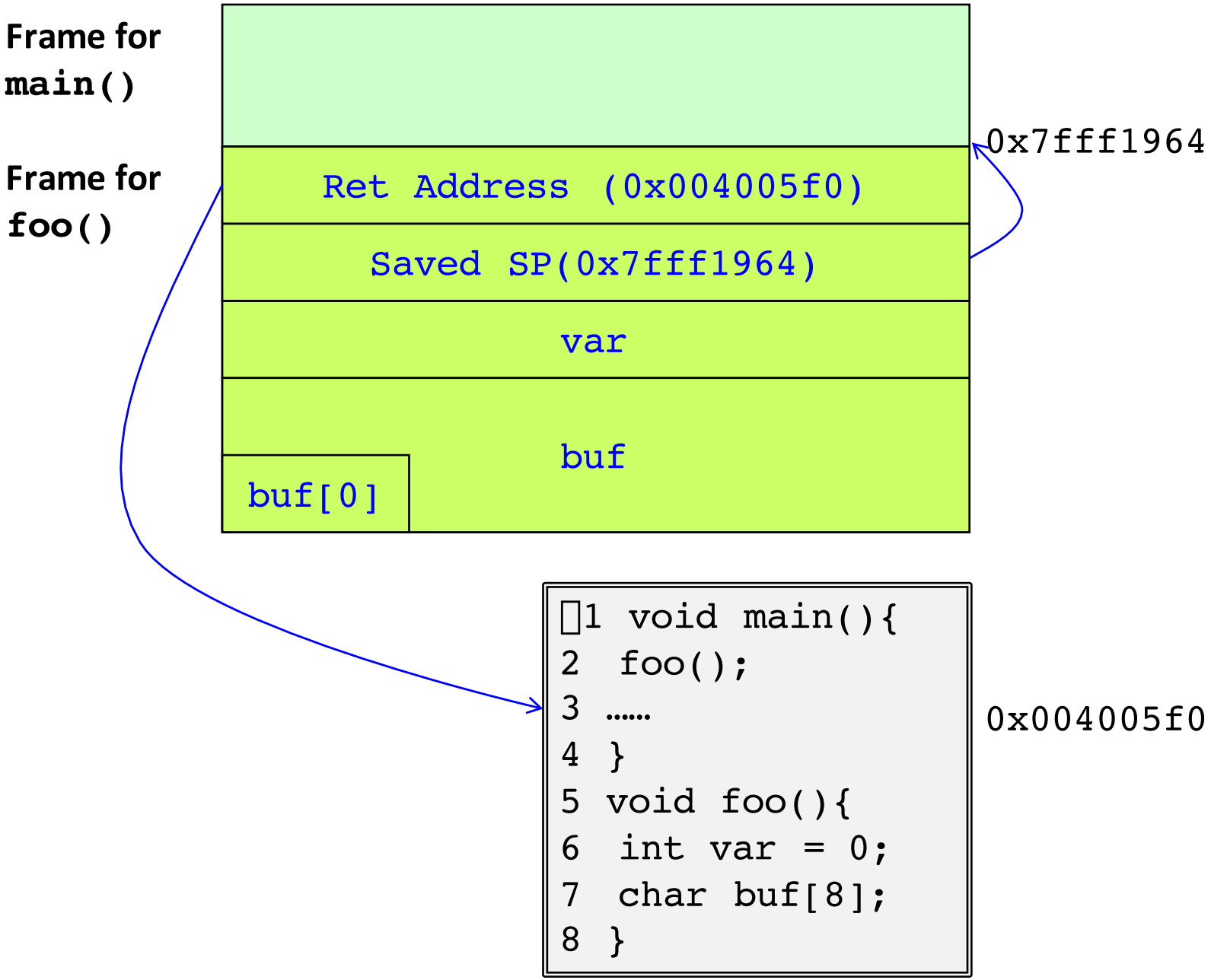
- Executable code
- Usually read-only

```
stack (lvar1) 0x7ffff8e4
stack (lvar2) 0x7ffff8d0
heap (d)      0x00d3f010
data (gvar)   0x00600aa0
data (s)      0x00600a60
text (foo)    0x0040054c
```

```c
#include <stdlib.h>
#include <stdio.h>
int gvar[1024];
void foo(){
  static int s = 5;
  int lvar1 = 1;
  char lvar2[10];
  void *d = malloc(512);
  printf("stack (lvar1)\t%014p\n", &lvar1);
  printf("stack (lvar2)\t%014p\n", lvar2);
  printf("heap (d)\t%014p\n", d);
  printf("data (gvar)\t%014p\n", gvar);
  printf("data (s) \t%014p\n", &s);
  printf("text (foo) \t%014p\n", foo);
}
void main(){
  foo();
}
```

Stack frame


Higher memory address
(e.g. 0xbfffffff)
Stack
Unused memory
Heap
Data
Text (Code)
Lower memory address
(e.g. 0x00000000)
Frame for main()
Frame for foo()
Stack Pointer
Instruction Pointer
1 void main(){
2 foo();
3 ……
4 }
5 void foo(){
6 ……
7 }


Frame for main()
Frame for foo()
Ret Address (0x004005f0)
Saved SP(0x7fff1964)
var
buf
buf[0]
0x7fff1964
1 void main(){
2 foo();
3 ……
4 }
5 void foo(){
6 int var = 0;
7 char buf[8];
8 }
0x004005f0

Buffer Overflow vulnerabilities

Buffer overflows

Writing data beyond a buffer's range

Common in programming languages with no built-in bounds checking like C & C++

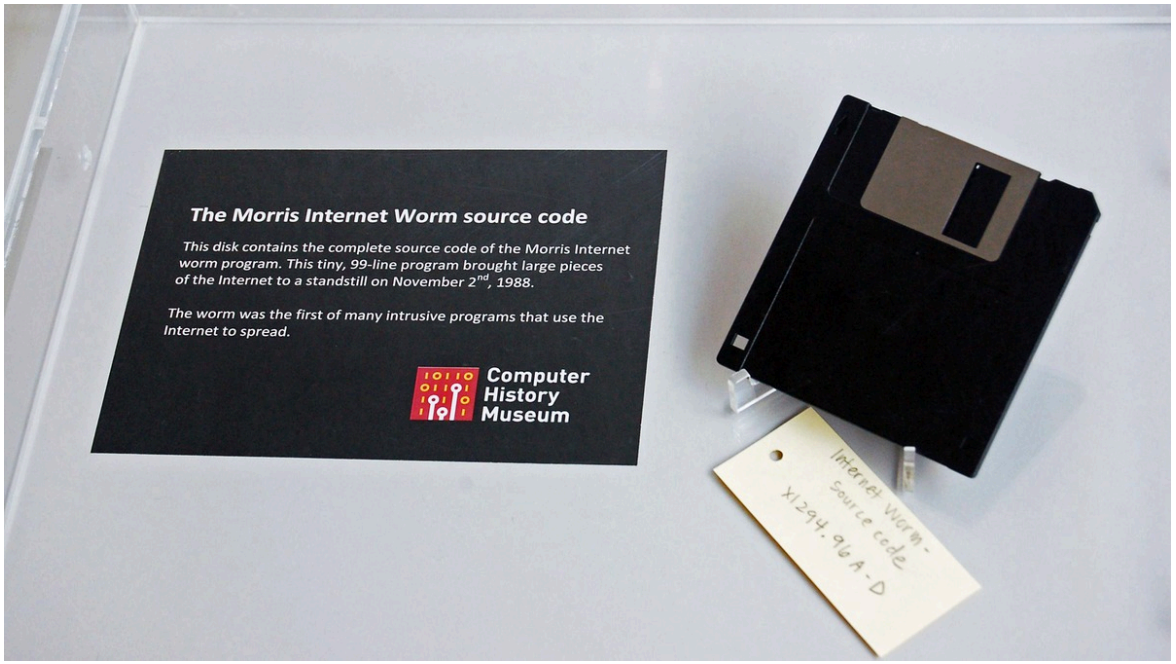
Can be exploited to corrupt the program's intended behavior

Example of buffer overflow exploits: Morris worm

Created and released by Robert Morris in 1988

One of the first Internet worms

Exploited buffer overflow in fingerd to self-replicate



Example of buffer overflow exploits: Morris worm

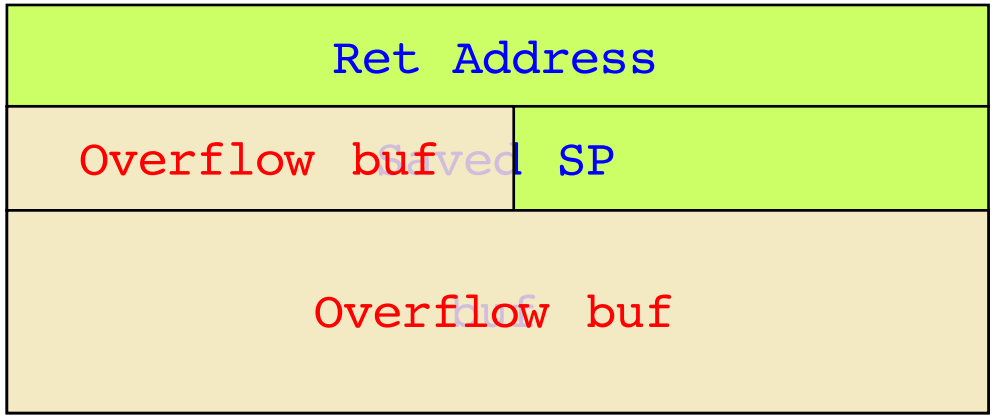
Spread faster than expected due to coding errors

Infected 6000 machines (10% back then!)

Consequence: sentenced to three years of probation, 400 hours of community service, and a fine of $10,050 plus the costs of his supervision

Morris is now a MIT professor

Frame for `vulnerable()`




```
void vulnerable() {
    char buf[8];
    gets(buf);
}
```

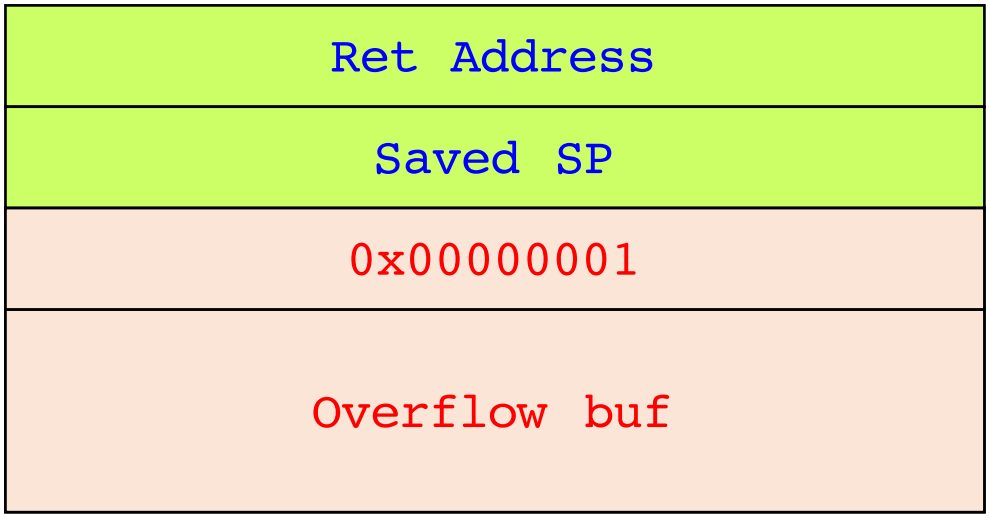
Feeding a string longer than 8 can crash the program, cause segmentation fault

Can it get worse?

```
void vulnerable() {
    int authenticated = 0;
    char buf[8];
    gets(buf);
    if (strncmp(buf, "s3cr3t",6))
        authenticated = 1;
    if (authenticated){
    /*grant access*/
    }
}
```

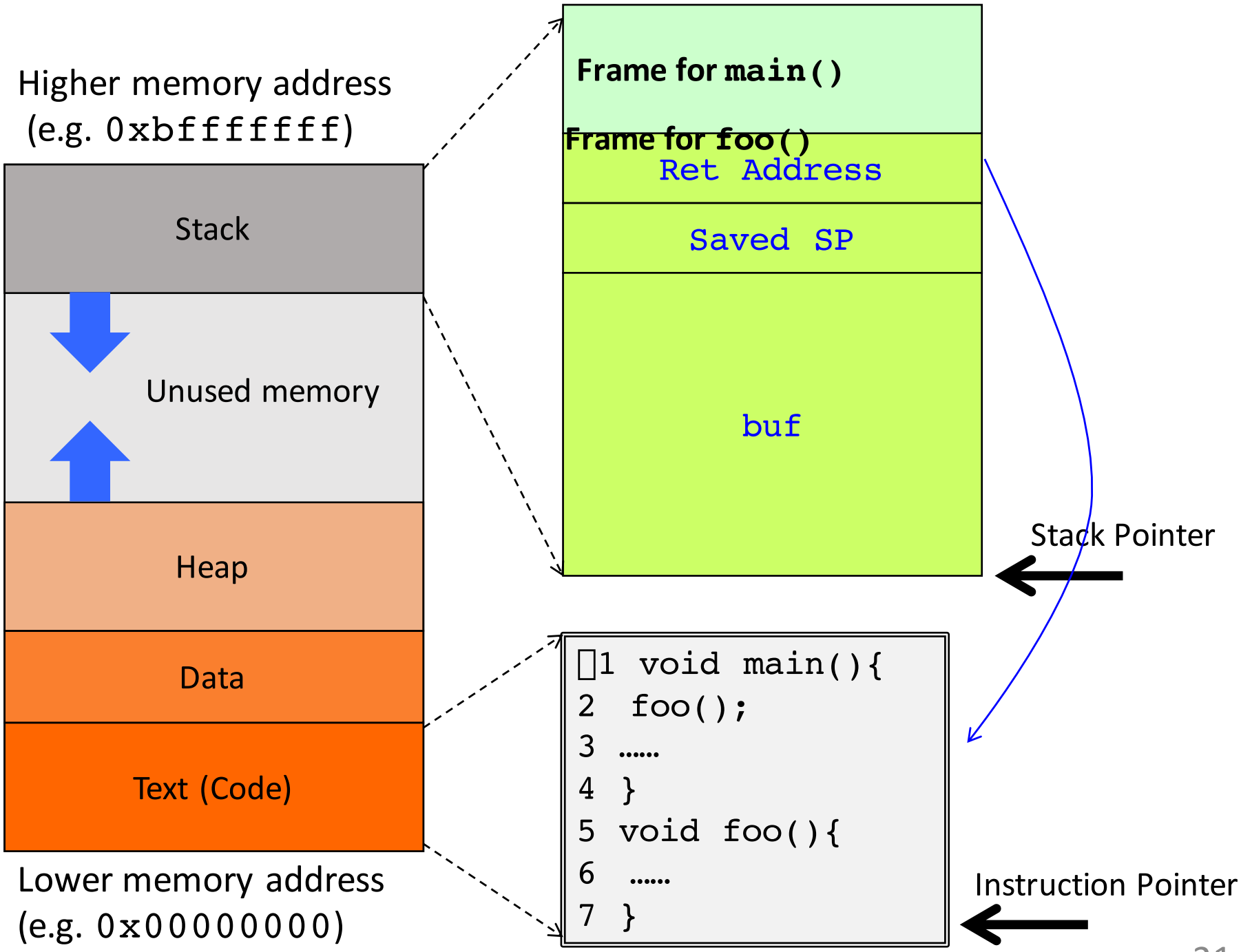
`Authenticated` is allocated next to `buf` and thus can be manipulated by an overrun `buf`

Frame for `vulnerable()`

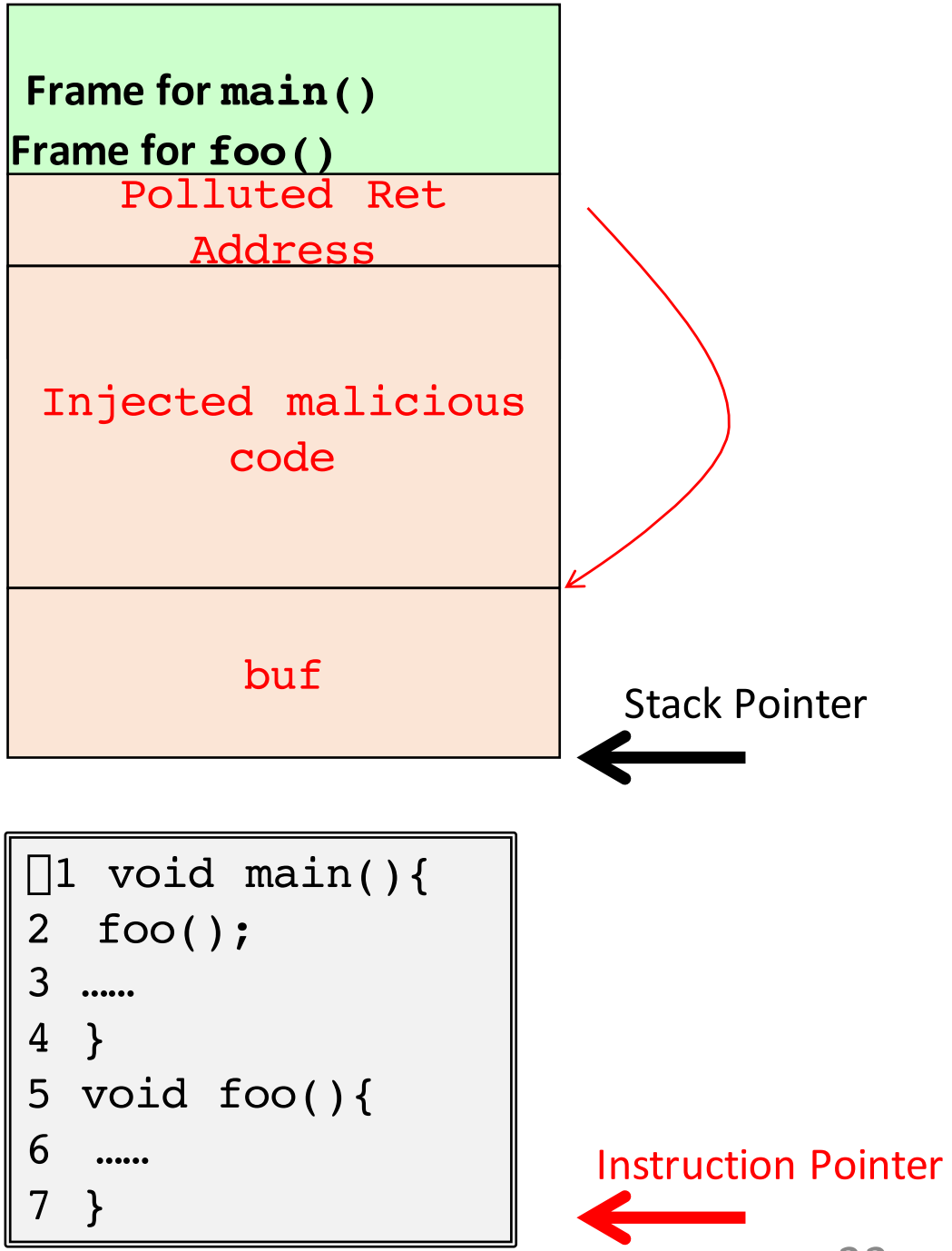


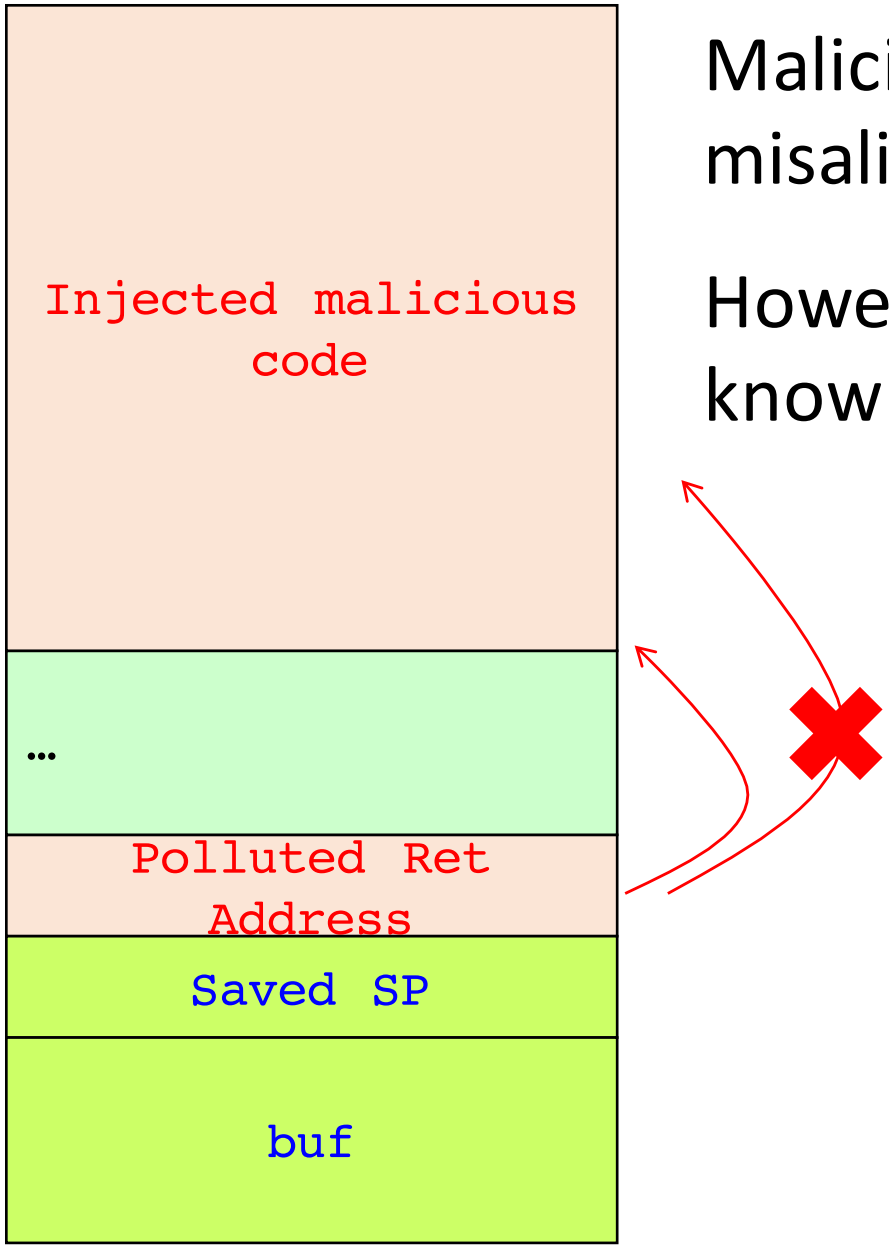
Can it get worse?

Can we take control and execute arbitrary malicious code?


Higher memory address
(e.g. 0xbfffffff)
Stack
Unused memory
Heap
Data
Text (Code)
Lower memory address
(e.g. 0x00000000)
Frame for main()
Frame for foo()
Ret Address
Saved SP
buf
Stack Pointer
1 void main(){
2 foo();
3 ……
4 }
5 void foo(){
6 ……
7 }
Instruction Pointer

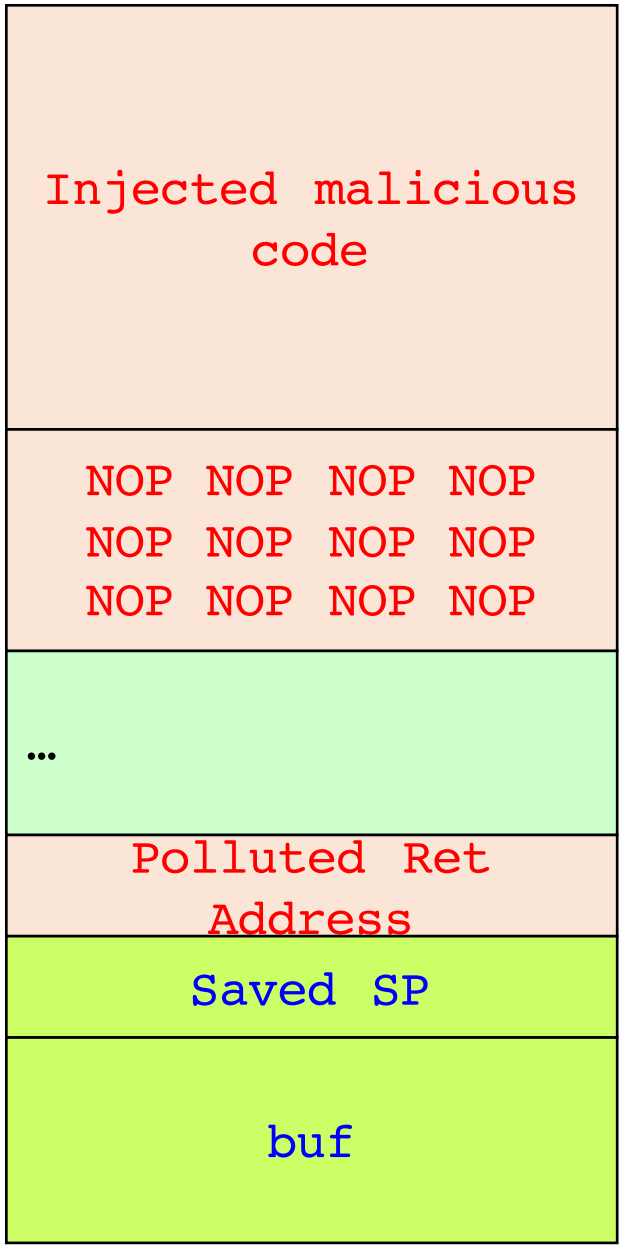
Control hijacking by polluting return address





Malicious code won't run even misaligned by only one byte

However, sometimes it's hard to know the exact offset



Prepend malicious code with NOPs (No Operation)

NOP simply advances Instruction Pointer by 1

=> Attacker only needs to know a rough address rather than an exact one!

Unsafe functions in standard C library

Function prototype	Potential problem
`strcpy(char *dest, const char *src)`	May overflow the **dest** buffer.
`strcat(char *dest, const char *src)`	May overflow the **dest** buffer.
`getwd(char *buf)`	May overflow the **buf** buffer.
`gets(char *s)`	May overflow the **s** buffer.
`fscanf(FILE *stream, const char *format, ...)`	May overflow its arguments.
`scanf(const char *format, ...)`	May overflow its arguments.
`realpath(char *path, char resolved_path[])`	May overflow the **path** buffer.
`sprintf(char *str, const char *format, ...)`	May overflow the **str** buffer.

Table 1: Partial List of Unsafe Functions in the Standard C Library

Safer versions with bounds checking

unsafe	safer
`strcpy(char*, const char*)`	`strncpy(char*, const char*, size_t)`
`strcmp(char*, const char*)`	`strncmp(char*, const char*, size_t)`
`gets(char*)`	`fgets(char*, int, FILE*)`
…	…

Exercise: how to skip “x=1”?

```
void function(int a, int b, int c) {
     char buffer1[5];
     char buffer2[10];
     int *ret;
     ret = buffer1 + ?; 8
     (*ret) += ?; 4
}
void main() {
     int x;
     x = 0;
     function(1,2,3);
     x = 1; /*skip me!*/
     printf("%d\n",x);
}
```

Alpha One. “Smashing The Stack For Fun And Profit”

Integer Conversion Vulnerabilities

PTT鄉民百科

本維基 | 鄉民中心 | 流行用語 | PTT名人 | 其他

維基動態 | 隨機頁面 | 影片 | 圖片 | 論壇

綠爆

473 個維基頁面

編輯 | 評論 0

綠爆，又稱作**淺綠爆**，是 ptt 看版人氣的一種指標，在單一看版人氣到達 30000 ~ 59999 人的時候，看版前面會顯示一個淺綠色的「爆」字。

歷史 編輯

2008年 編輯

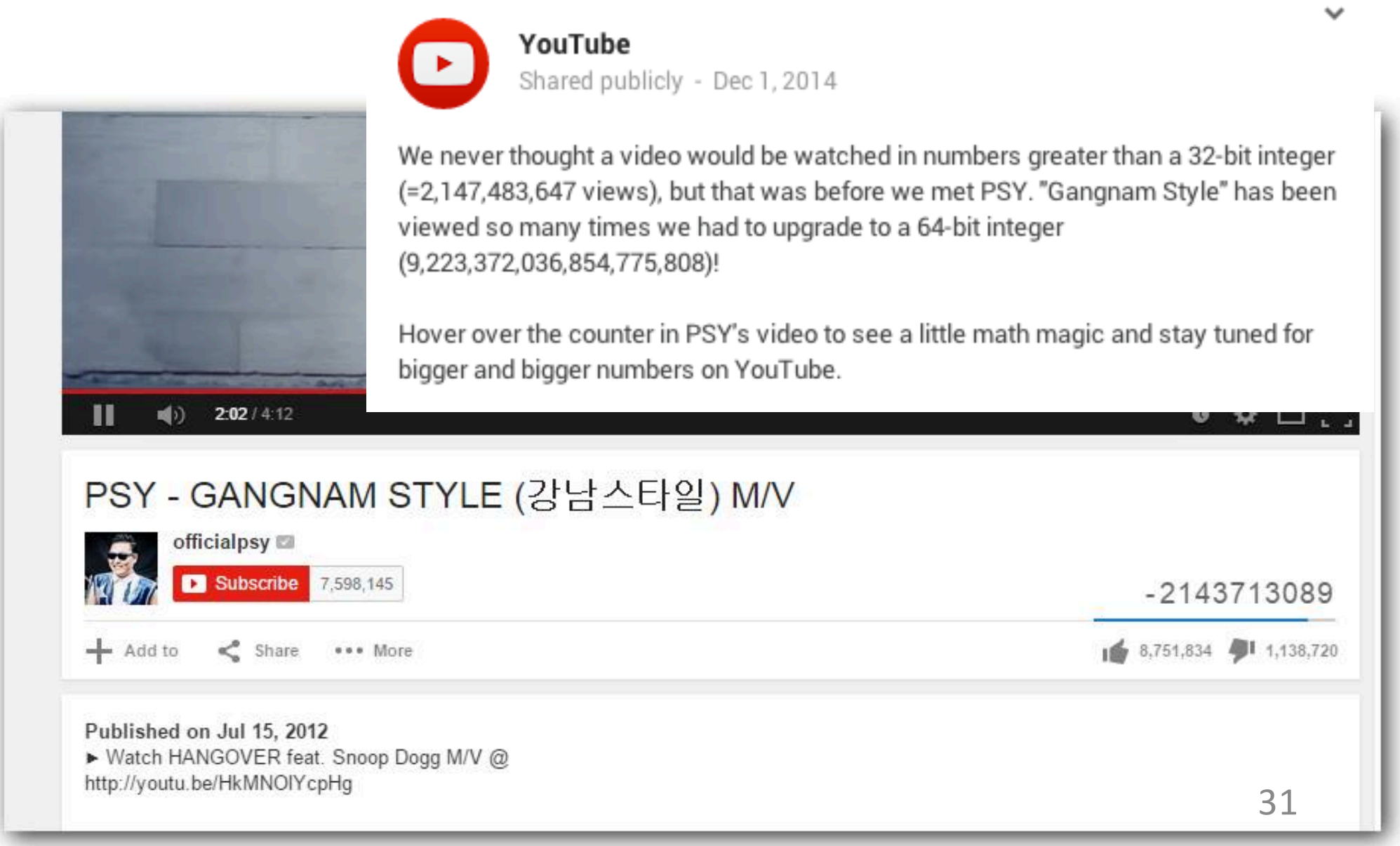
八搶三奧運 編輯

在 2008 年 3 月 10 日，棒球版在八搶三奧運資格中華隊對加拿大的比賽時，因戰況激烈互有領先，且羅國輝擊出三分全壘打，人氣終於突破 3 萬人，達成史上第一次綠爆，人氣並持續上昇，但到了 32767 人時，系統突然出問題，右上角人氣顯示開始變負數，而版前的爆不見變成黑洞，所以後來無法確定這次最高人數，有人說可能在 4 萬人左右。後來 ptt 經過這次後就把該系統問題修好了。

兩天後也就是 3 月 12 日，中華隊對上澳洲隊並獲勝，確定進入北京奧運，再度綠爆（這次系統沒問題了），有人（carhow大）錄得最高人氣 37470。

Integer overflows（溢位）

Integer overflows （溢位）



```
void vulnerable() {
    size_t len;
    char *buf;
    len = read_int_from_network();
    buf = malloc(len+5);
    read(fd, buf, len);
    ...
}
```

輸入一個很大的len，讓他加5後就overflow

重點是要讓len bypass 長度檢查，變超大

```
int vulnerable(char* buf1, char* buf2,
    int len1, int len2) {

    char concat[1024];
    if((len1 + len2) > 1024)
        return -1;
    memcpy(temp, buf1, len1);
    memcpy(temp+len1, buf2, len2);
}
```

Conversion problem

```
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > 80) {
        error("length too long");
        return;
    }
    memcpy(buf, p, len);
}
```

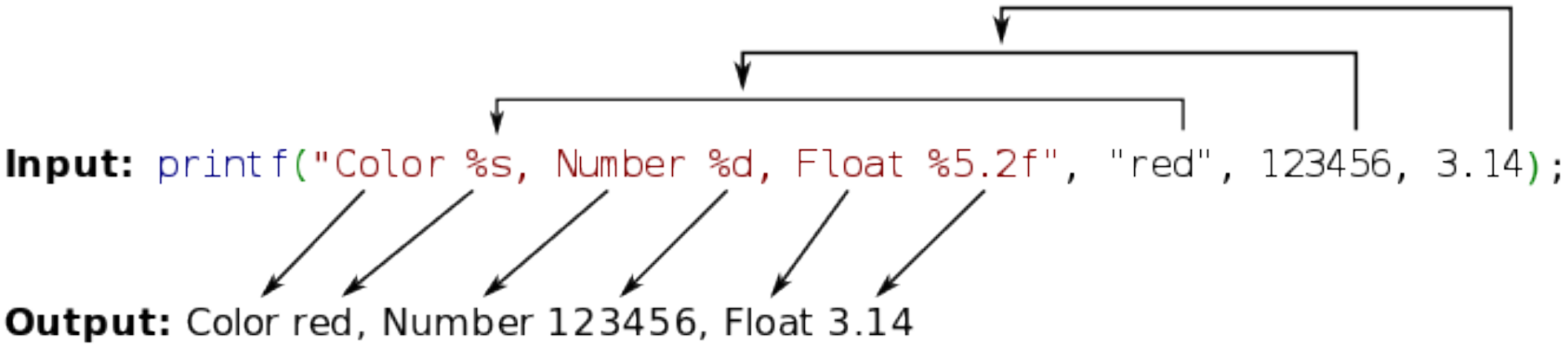
會被當unsign

Hint:

```
void *memcpy(void *dest, const void *src, size_t n);
typedef unsigned int size_t;
```

Format String Vulnerabilities

Format string



Functions using format string:

- printf, fprintf, sprintf, vprintf, vfprintf, vsprintf, syslog, err, warn…

Example: printf

```
int printf(const char* format, …);
```

- Writes the C string pointed by format to the standard output (stdout). If format includes format specifiers (subsequences beginning with %), the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers. -- http://www.cplusplus.com/reference/cstdio/printf/

Proper usage:

- `printf("Hello World!");`
- `printf("%s", buffer);`
- `printf("Name:%s Age:%d", name, age);`

```
void vulnerable() {
    char buf[80];
    if (fgets(buf, sizeof buf, stdin) == NULL)
        return;
    printf(buf);
}
```

`buf` may contain format specifiers supplied by the user!

Example: printf

```
int printf(const char* format, …);
```

Proper usage:

- `printf("Hello World!");`
- `printf("%s", buffer);`
- `printf("Name:%s Age:%d", name, age);`

Bad usage:

- `printf(buffer);`

Format string vulnerabilities

Print memory content (or crash):

- `printf("%s%s%s%s");`

Print the first two words of memory in hex:

- `printf("%x%x");`

Exploting Format String Vulnerabilities
https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf

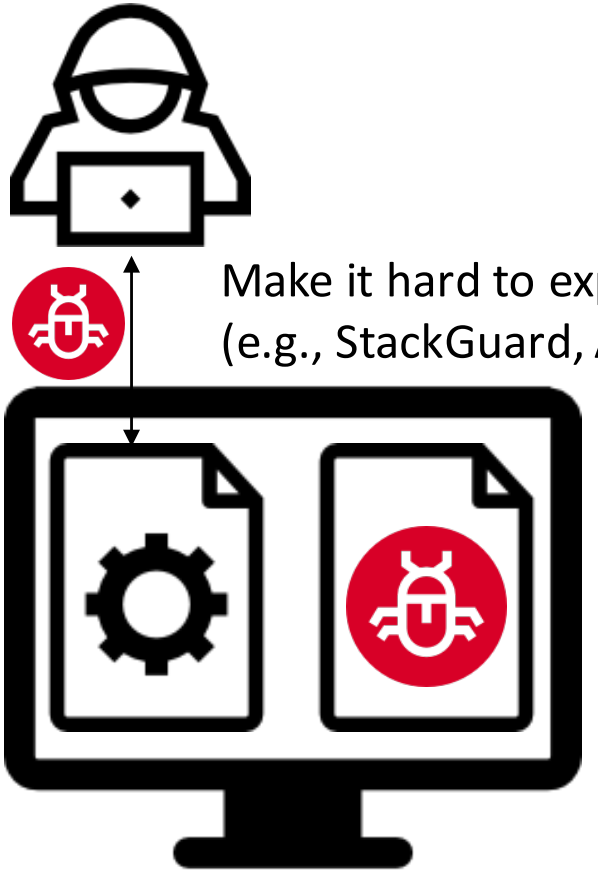
Format string vulnerabilities

%n: Nothing printed. The number of characters written so far is stored in the pointed location.

Write any value to any address:

- `printf("Hello %n", &pos);`
  - /* `pos` becomes 6*/

Defending against memory-safety vulnerabilities



Make it hard to exploit
(e.g., StackGuard, ASLR)

Eliminate bugs
(e.g., type-safe language, formal methods, software testing)

Prevent malware injection/ download
(e.g., firewall, WAF)

Detect execution of malicious code
(e.g., antivirus, behavior monitoring)

Ensure running only trusted code
(e.g., attestation, trusted computing)

* These approaches complement each other

Example: Secure coding

Defensive programming: don't trust any other functions!

- 就如同**防衛性駕駛**：不要假設其他駕駛或行人會看路！

Always perform checks even seems redundant

- Bounds checking to prevent buffer overflows and integer overflows/underflows
- Check if memory is initialized before use
- Check if memory has been freed before freeing it
- Check if a pointer is not NULL before dereference

Example: Better languages and libraries

Unsafe: C, C++, assembly, …

Safer: Java, Python, Ruby, …

- Automatically perform bounds checking
- Automatically resize buffer
- No easy access to memory address
- Strong typed

Disadvantages: legacy code, performance

Example: Runtime checking

Add runtime code to detect exploits dont trust library

- E.g. enhance C compiler to add bounds checking at every array or pointer access

Throw exceptions when exploits detected

- StackGuard, LibSafe, …

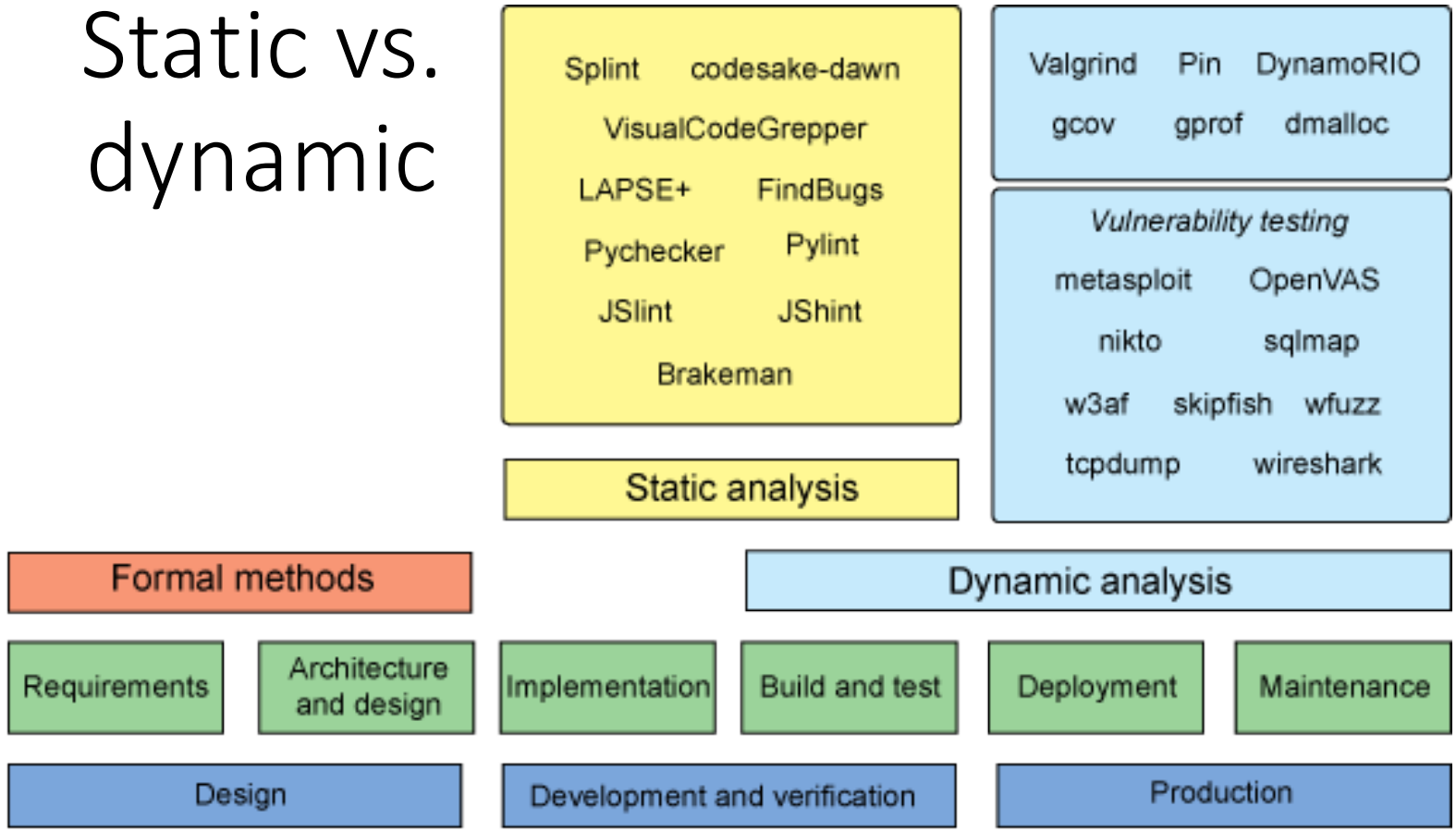
Disadvantages: Legacy code, performance

Automated Analysis

Static analysis: Find bugs at compile time (without actually running it)

Dynamic analysis: Examination of a program during run time

Static vs. dynamic

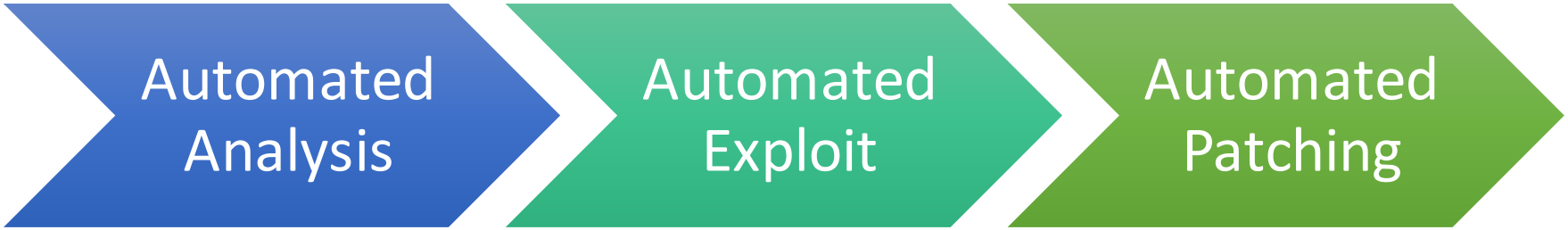


http://www.ibm.com/developerworks/library/se-static/

Why using automated techniques?

Manual inspection cannot catch up with the growing scale and complexity of software systems.

Grand Research Challenge: machines automatically discover, exploit, patch new vulnerabilities.



DARPA Challenges

2005 driverless Car Challenge

2007 Urban Challenge

2012 Robotics Challenge

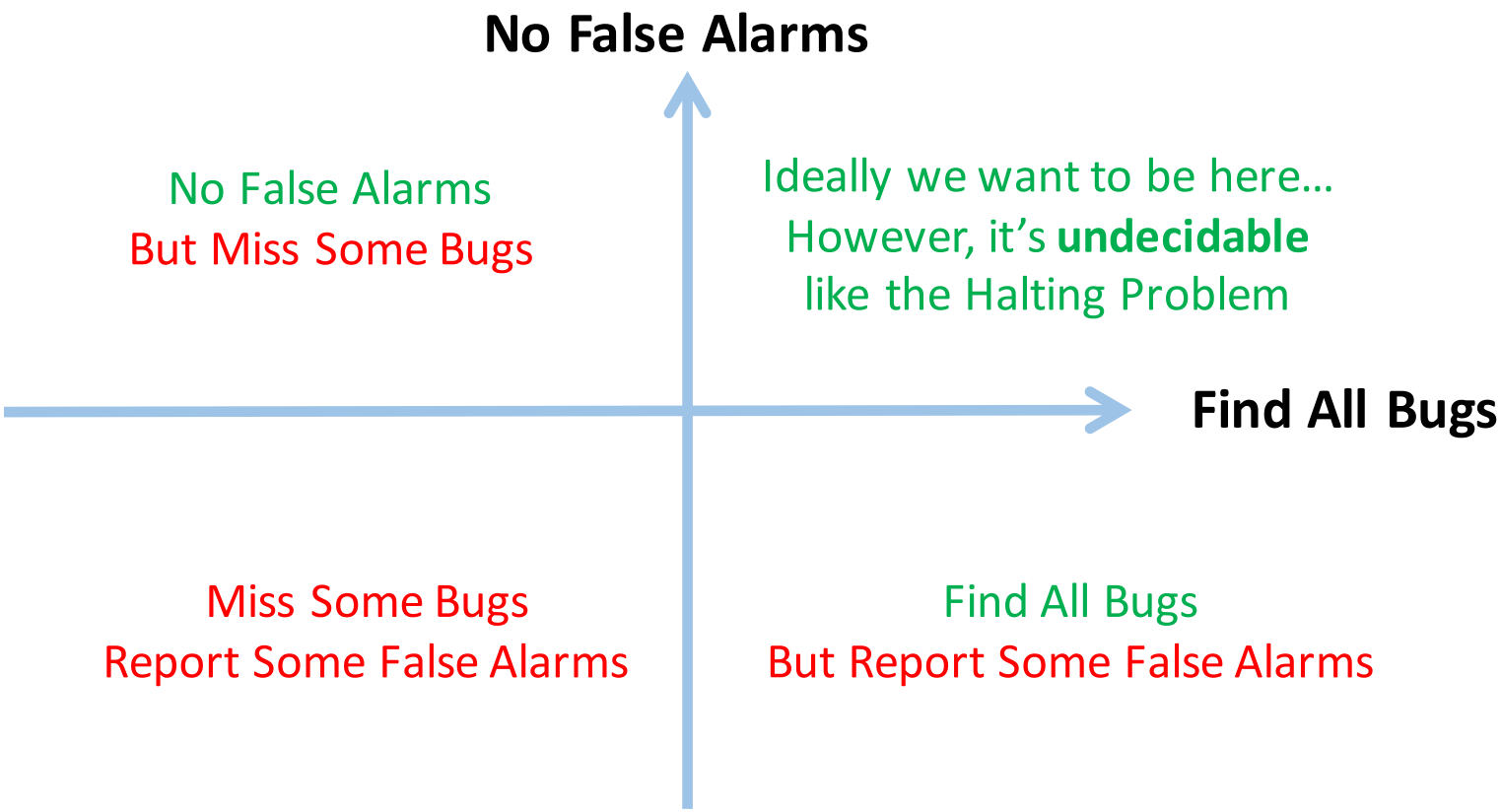
DARPA Cyber Grand Challenge (CGC) in 2016

All Machine Hacking Tournament

Automatic attack and defense

Finalists: 7 teams including UC Berkeley, CMU, UCSB, …

Automated Program Analysis

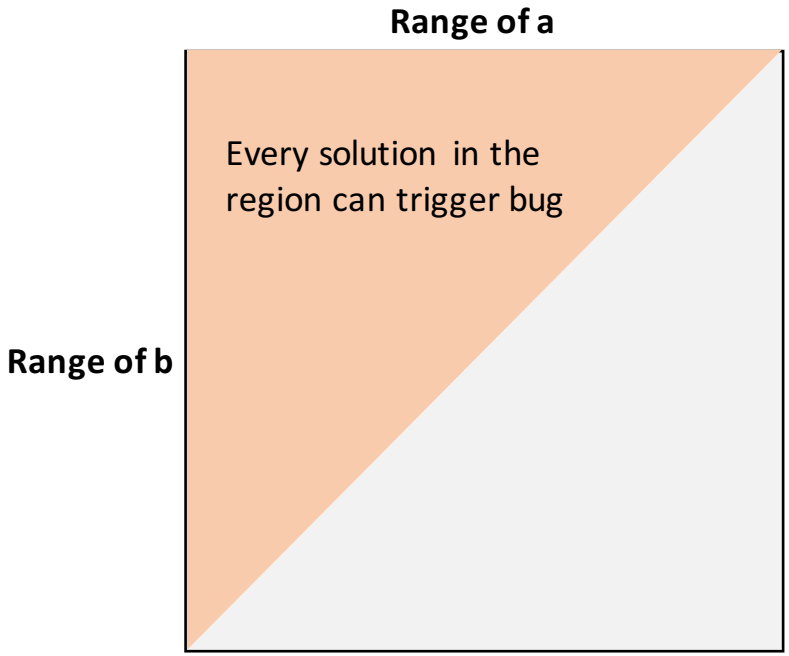


Core Concept of Automated Program Analysis

Examines possible values and see whether any one leads to crash

```
void foo(int a, int b) {
  if (a > b){
    happy();
  }
  else {
    buggy();
  }
}
```

Input to trigger this?

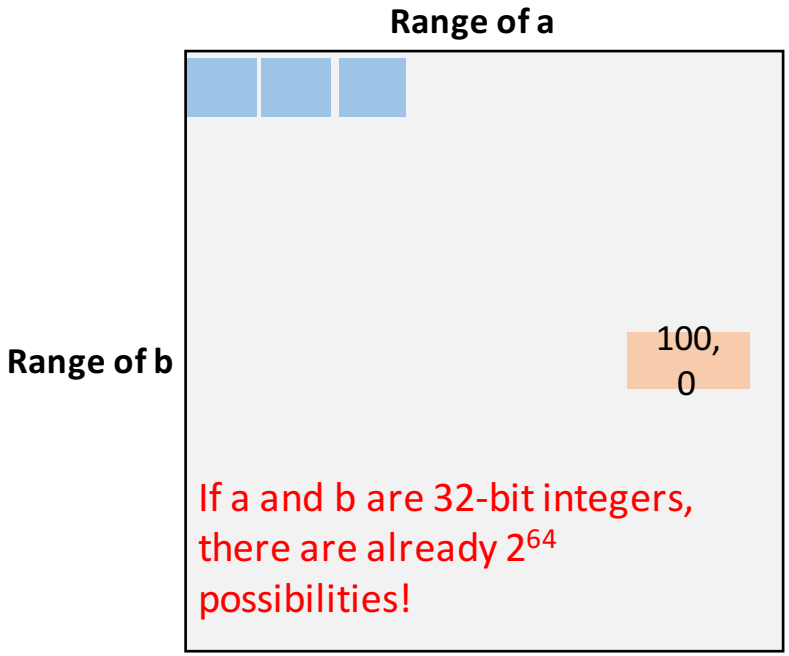


Core Concept of Automated Program Analysis

Examines possible values and see whether any one leads to crash

```
void foo(int a, int b) {
  if (a != 100){
    happy();
  }
  else {
    if (b == 0) {
      buggy();
    }
  }
}
```

Input to trigger this?



Core Concept of Automated Program Analysis

Examines possible values and see whether any one leads to crash

Impractical to exhaustively try every possible value

- Two 32-bit integers -> 264 possibilities!

Automated program analysis techniques are about how to intelligently explore the input space so that we can cover more "interesting" results in bounded computation/time

Fuzzing and **symbolic execution** are two popular techniques but each has its advantages and limitations

Fuzzing (模糊測試)

一種實務上被廣泛使用之自動化分析漏洞的技術

核心概念：隨機地生成輸入，試試看是否能觸發漏洞(異常結束)。

通常會藉由某些啟發式策略，如覆蓋率導向，排序尚待測試的輸入值。

Fuzzing (模糊測試)

Provide large amount of random, unexpected data as input of a program; monitor and check whether exception events occur.

- E.g., program crashes or built-in assertion fails.

Advantages

- Fast, simple testing design
- Free of preconceptions about internal implementations

Disadvantages

- Hard to find sophisticated vulnerabilities.
- E.g., a if-else statement like this: "if (x == 5566)"

大於或小於就可能，等於就不合適

Fuzzing (模糊測試)

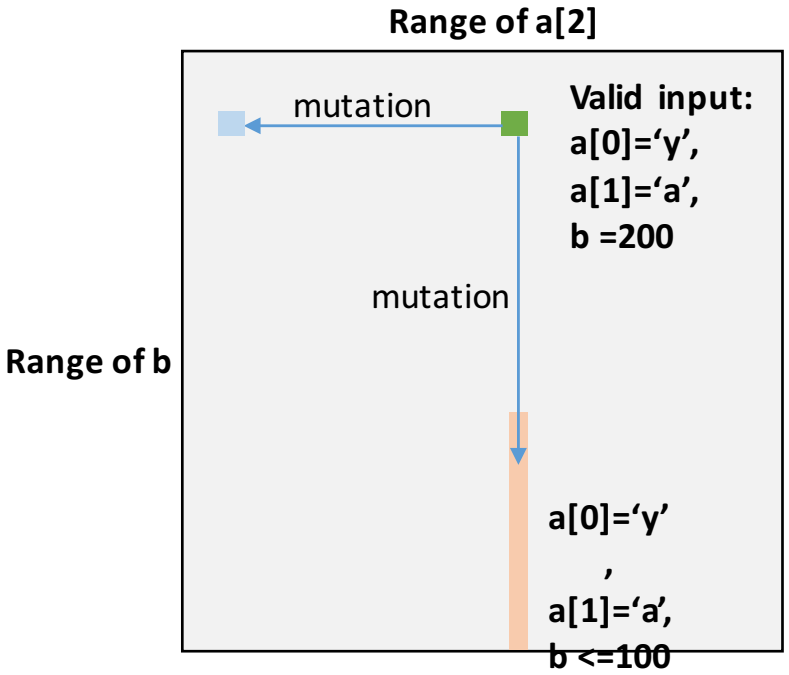
Current Status: More and more popular, research topics including:

- Focus on how to fuzz wisely.
  - Mutation-based Fuzzer
  - Generation-based Fuzzer
- Focus on how to improve performance of fuzzing
  - Lightweight Instrumentation for Coverage-guided Fuzzing (Our work)
- Focus on how to fuzz specific part of program. (e.g., patch)
  - Directed Fuzzer (AFL-go, AFL-fast)
- Combine with symbolic execution.
  - Help to bypass difficult if-else statement more efficiently. (Driller)

Mutational fuzzing

Given valid inputs and mutate them based on some heuristics (e.g., bit flipping)

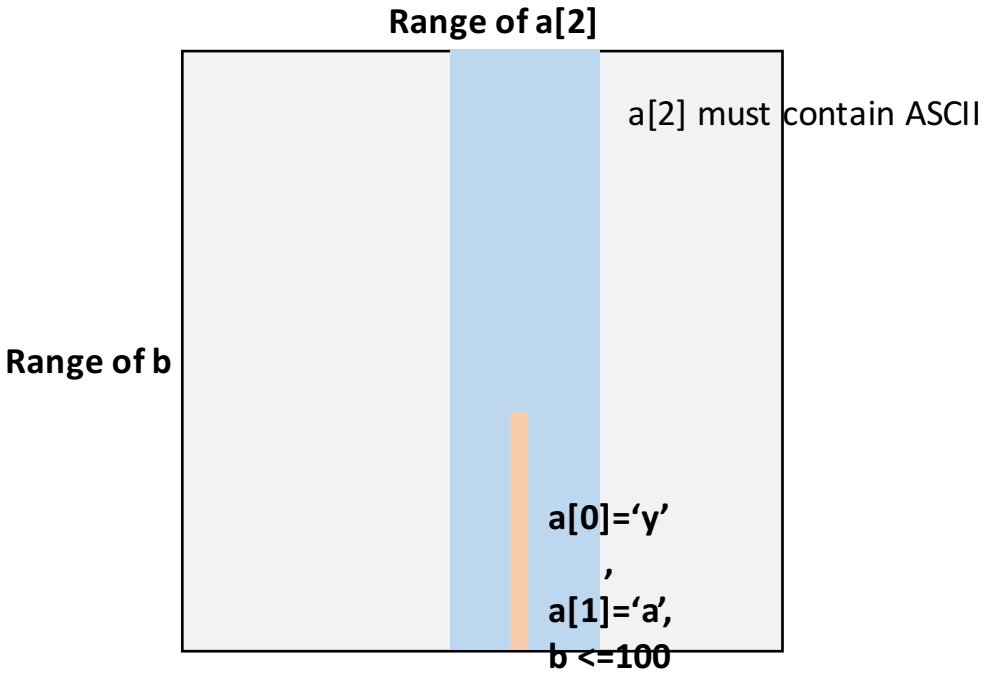
```
void foo(int a[], int b) {
  if (a[0] != 'y' || a[1] != 'a') {
    return;
  }
  if (b > 100){
    happy();
  }
  else {
    buggy();
  }
}
```



Generational fuzzing

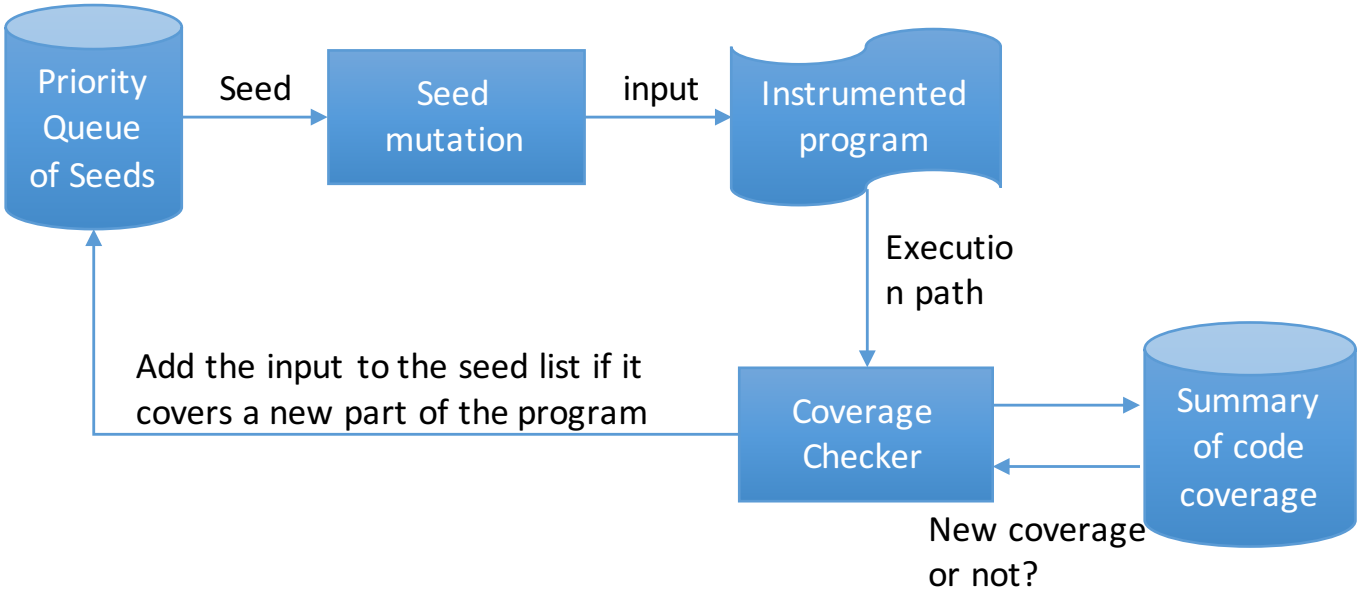
Given input specification to reduce the search space

```
void foo(int a[], int b) {
  if (a[0] != 'y' || a[1] != 'a') {
    return;
  }
  if (b > 100){
    happy();
  }
  else {
    buggy();
  }
}
```



Coverage-guided mutational Fuzzing

This feedback strategy has shown to be effective in finding real-world vulnerabilities (e.g., AFL).



File Format Fuzzers

Fuzzers which helps in fuzzing file formats like pdf, mp3, swf etc.,

MiniFuzz - Wayback Machine link - Basic file format fuzzing tool by Microsoft. (No longer available on Microsoft website).

BFF from CERT - Basic Fuzzing Framework for file formats.

AFL Fuzzer (Linux only) - American Fuzzy Lop Fuzzer by Michal Zalewski aka lcamtuf

Win AFL - A fork of AFL for fuzzing Windows binaries by Ivan Fratic

Shellphish Fuzzer - A Python interface to AFL, allowing for easy injection of testcases and other functionality.

TriforceAFL - A modified version of AFL that supports fuzzing for applications whose source code not available.

Peach Fuzzer - Framework which helps to create custom dumb and smart fuzzers.

MozPeach - A fork of peach 2.7 by Mozilla Security.

Failure Observation Engine (FOE) - mutational file-based fuzz testing tool for windows applications.

rmadair - mutation based file fuzzer that uses PyDBG to monitor for signals of interest.

honggfuzz - A general-purpose, easy-to-use fuzzer with interesting analysis options. Supports feedback-driven fuzzing based on code coverage. Supports GNU/Linux, FreeBSD, Mac OSX and Android.

zzuf - A transparent application input fuzzer. It works by intercepting file operations and changing random bits in the program's input.

radamsa - A general purpose fuzzer and test case generator.

binspector - A binary format analysis and fuzzing tool

grammarinator - Fuzzing tool for file formats based on ANTLR v4 grammars (lots of grammars already available from the ANTLR project).

Network Protocol Fuzzers

Fuzzers which helps in fuzzing applications which use network based protocals like HTTP, SSH, SMTP etc.,

Peach Fuzzer - Framework which helps to create custom dumb and smart fuzzers.

Sulley - A fuzzer development and fuzz testing framework consisting of multiple extensible components by Pedram Amini.

boofuzz - A fork and successor of Sulley framework.

Spike - A fuzzer development framework like sulley, a predecessor of sulley.

Metasploit Framework - A framework which contains some fuzzing capabilities via Auxiliary modules.

Nightmare - A distributed fuzzing testing suite with web administration, supports fuzzing using network protocols.

rage_fuzzer - A dumb protocol-unaware packet fuzzer/replayer.

Misc

Other notable fuzzers like Kernel Fuzzers, general purpose fuzzer etc.,

KernelFuzzer - Cross Platform Kernel Fuzzer Framework.

honggfuzz - A general-purpose, easy-to-use fuzzer with interesting analysis options.

Hodor Fuzzer - Yet Another general purpose fuzzer.

libFuzzer - In-process, coverage-guided, evolutionary fuzzing engine for targets written in C/C++.

syzkaller - Distributed, unsupervised, coverage-guided Linux syscall fuzzer.

ansvif - An advanced cross platform fuzzing framework designed to find vulnerabilities in C/C++ code.

Fuzzing Tools

https://github.com/secfigo/Awesome-Fuzzing

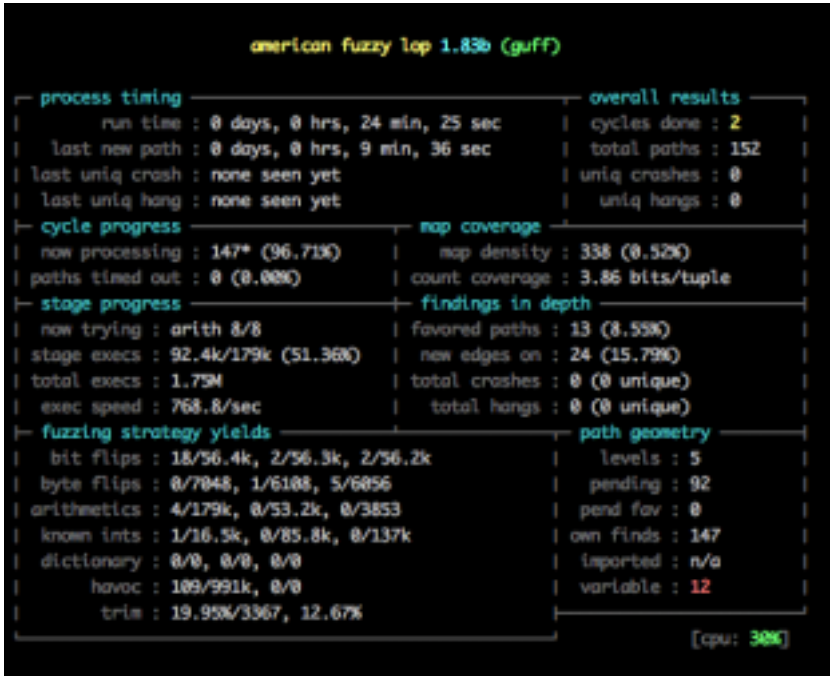
American Fuzzy Loop (AFL)

American Fuzzy Loop (AFL)為目前最廣為使用之模糊測試工具

至少四隊CGC隊伍使用AFL

AFL能對原始碼和（透過QEMU）執行檔進行instrumentation，藉此收集分支覆蓋（branch coverage）的資訊。

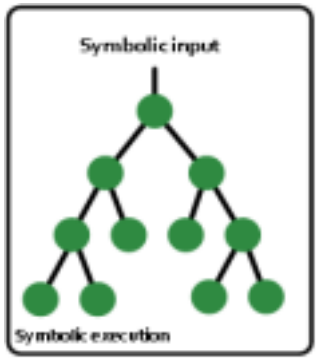
AFL使用演化演算法反饋輸入值的成效，在測試期間會偏好保留有提升覆蓋率的輸入，並進一步突變。

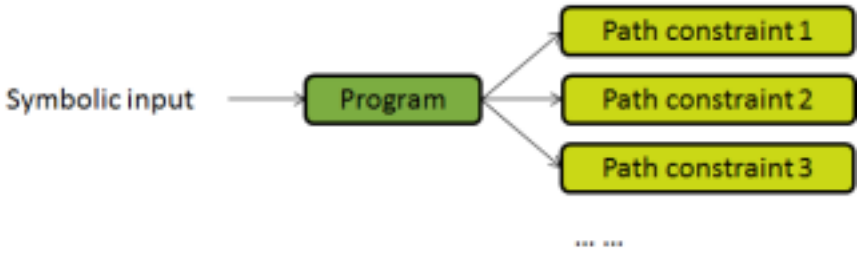


Symbolic Execution (符號執行)

A program analysis technique that systematically explores a program & generate test inputs to trigger bugs

- Inputs are represented as symbolic values 類似tensorflow的graph
- Execution paths are represented as logical formula (i.e., first-order logic constraints over symbolic values) called path constraints
- Concrete input values that execute a path can be found by solving path constraints using an SMT solver





Symbolic Execution

常見應用：軟體測試和脅迫生成

根據不同的程式語言以及需求，現今已發展出許多不同的符號執行平台，如Java PathFinder, KLEE, angr, TRITON



Concrete vs. Symbolic Execution

Concrete execution:
用實際的輸入值執行程式，一次探索一條執行路徑

Symbolic execution:
用符號值執行程式，相當於同時探索多個執行路徑

Concolic execution:

- Concrete + Symbolic
- 同時保留concrete的值和symbolic的表示式
- 可保有兩者的優點：如先用concrete execution走過一條路徑，再用symbolic execution收集這條路徑上的constraints；遇到不知如何symbolically處理的操作，就使用concrete值處理。

Symbolic Execution

SE process can be simplified into two phases:

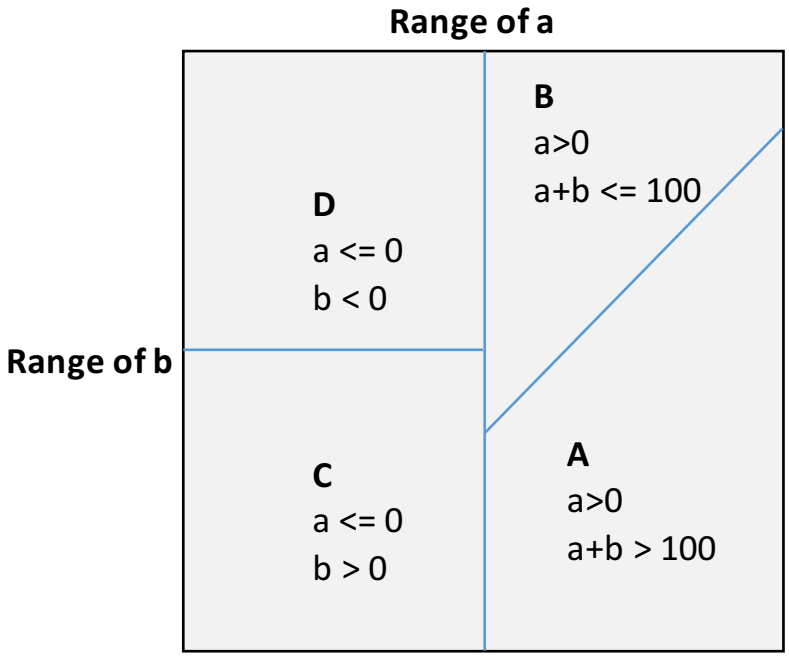
- Path Exploration
  - Execute the target program with symbolic input variables to explore possible execution path. Usually a explore strategy is utilized.
- Constraint solving
  - Solving a path constraint to obtain the concrete input values that cause the program execute along given path. Usually a SMT solver is utilized.

These two phases can interleave, depending on the design of symbolic execution engines.

Symbolic Execution (符號執行)

Reason on classes of inputs rather than single inputs

```
void foo(int a, int b) {
  if (a > 0){
    if (a + b > 100) {
      ...          A
    }
    else {
      buggy();     B
    }
  }
  else {
    if (b > 0) {
      ...          C
    }
    else {
      ...          D
    }
}
```



2^{64} possibilities --> 4 possibilities!

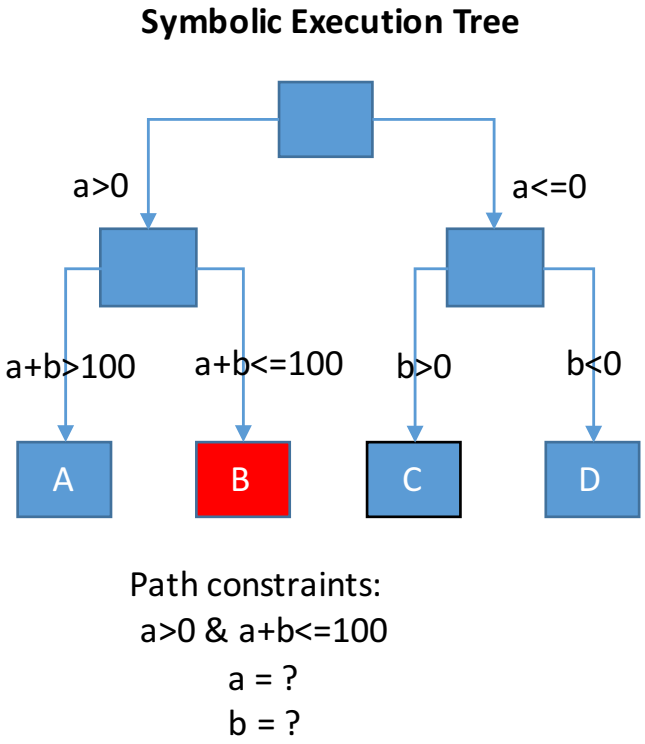
SMT Solvers

Satisfiability modulo theories (SMT) generalizes the SAT problem

- SAT: Satisfiability of Boolean formula

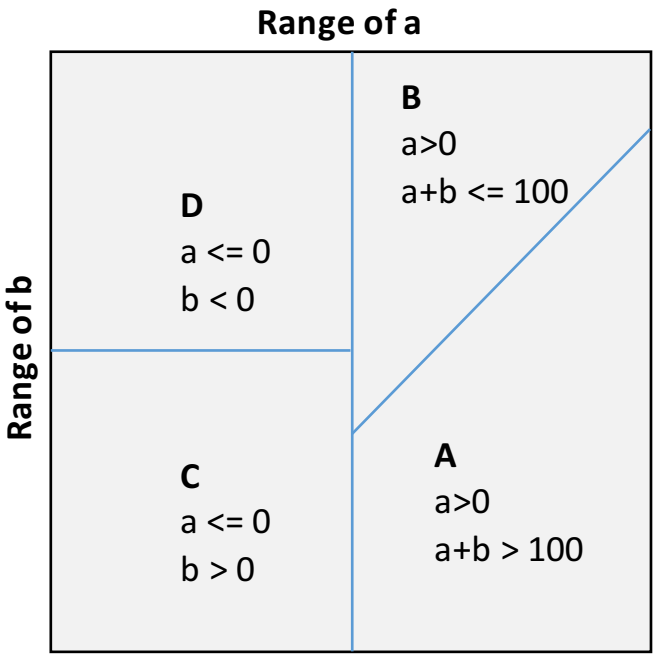
Checking the feasibility of a path

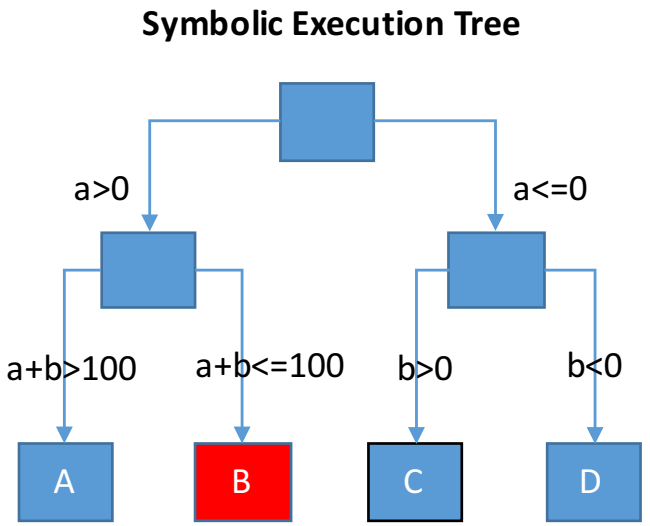
Generating assignments to symbolic variables



Symbolic Execution (符號執行)

```
void foo(int a, int b) {
  if (a > 0){
    if (a + b > 100) {
      ...          A
    }
    else {
      buggy();     B
    }
  }
  else {
    if (b > 0) {
      ...          C
    }
    else {
      ...          D
    }
}
```





Path constraints:
a>0 & a+b<=100
a = ?
b = ?

Challenges of Practical Symbolic Execution

SE has been around for decades but is still far from practical

Main technical challenges

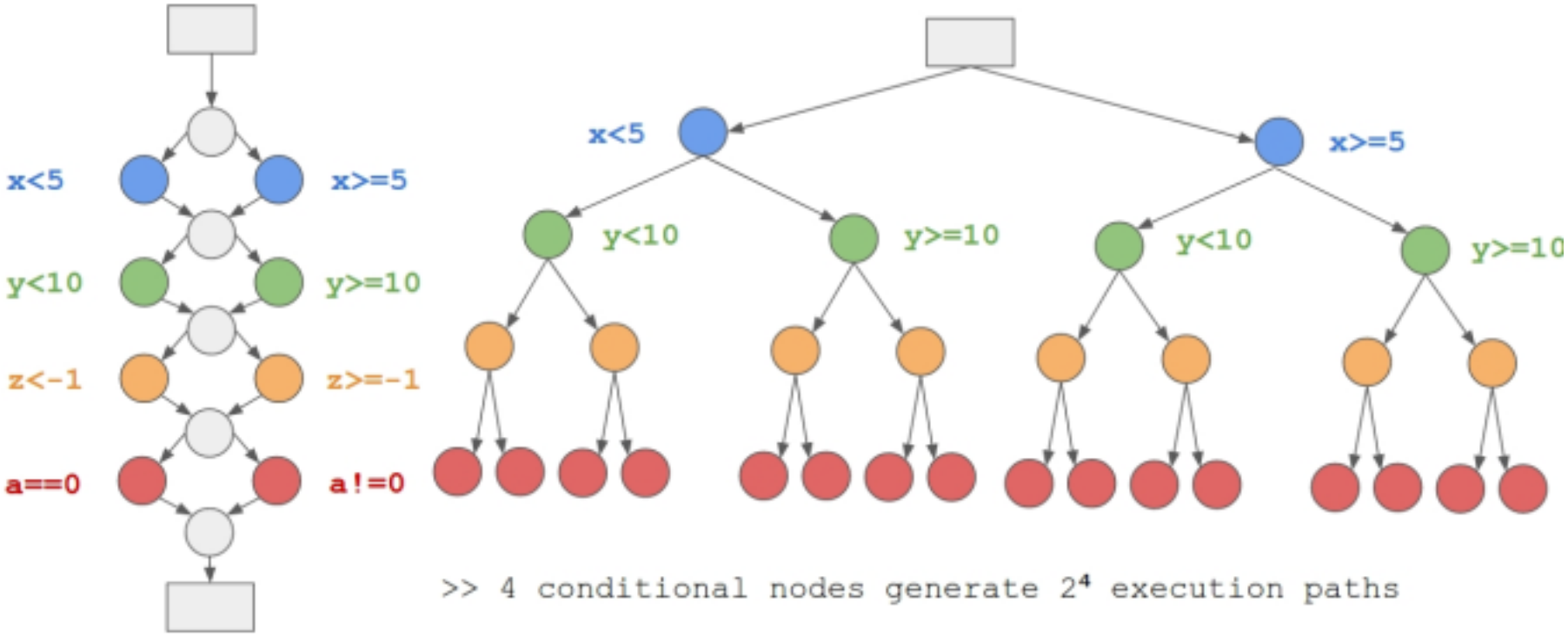
- State explosion
- Expensive constraint solving

Scalability issues

- Environment modeling
- Memory handling

Modeling issues

State Explosion



Handling State Explosion

Common idea: only explore a subset of "interesting" state and prioritize them

- May have false negatives

Approaches

- Pruning unsatisfiable paths
- State Merging
- Search heuristics
- Concolic execution
- Under-constrained symbolic execution
- …

Handling State Explosion - Search Heuristics

決定探索Symbolic Execution Tree的順序

各有優缺點

Heuristic	Goal
BFS	*Maximize coverage* [Chipounov et al., 2012, Tillmann and De Halleux, 2008]
DFS	*Exhaust paths, minimize memory usage* [Cadar et al., 2006, Chipounov et al., 2012] [Tillmann and De Halleux, 2008, Godefroid et al., 2005]
Random path selection	*Randomly pick a path with probability based on its length* [Cadar et al., 2008]
Code coverage search	*Prioritize paths that may explore unexplored code* [Cadar et al., 2006, Cadar et al., 2008, Cha et al., 2012] [Chipounov et al., 2012, Groce and Visser, 2002]
Buggy-path-first	*Prioritize bug-friendly path* [Avgerinos et al., 2011]
Loop exhaustion	*Fully explore specific loops* [Avgerinos et al., 2011]
Symbolic instruction pointers	*Prioritize paths with symbolic instruction pointers* [Cha et al., 2012]
Symbolic memory accesses	*Prioritize paths with symbolic memory accesses* [Cha et al., 2012]
Fitness function	*Prioritize paths based on a fitness function* [Xie et al., 2009, Cadar and Sen, 2013, Xie et al., 2009]
Subpath-guided search	*Use frequency distributions of explored subpaths to prioritize less covered parts of a program* [Li et al., 2013]

Figure 10: Common path selection heuristics discussed in literature.

Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., & Finocchi, I. (2016). *A Survey of Symbolic Execution Techniques*.

Handling State Explosion - Concolic execution

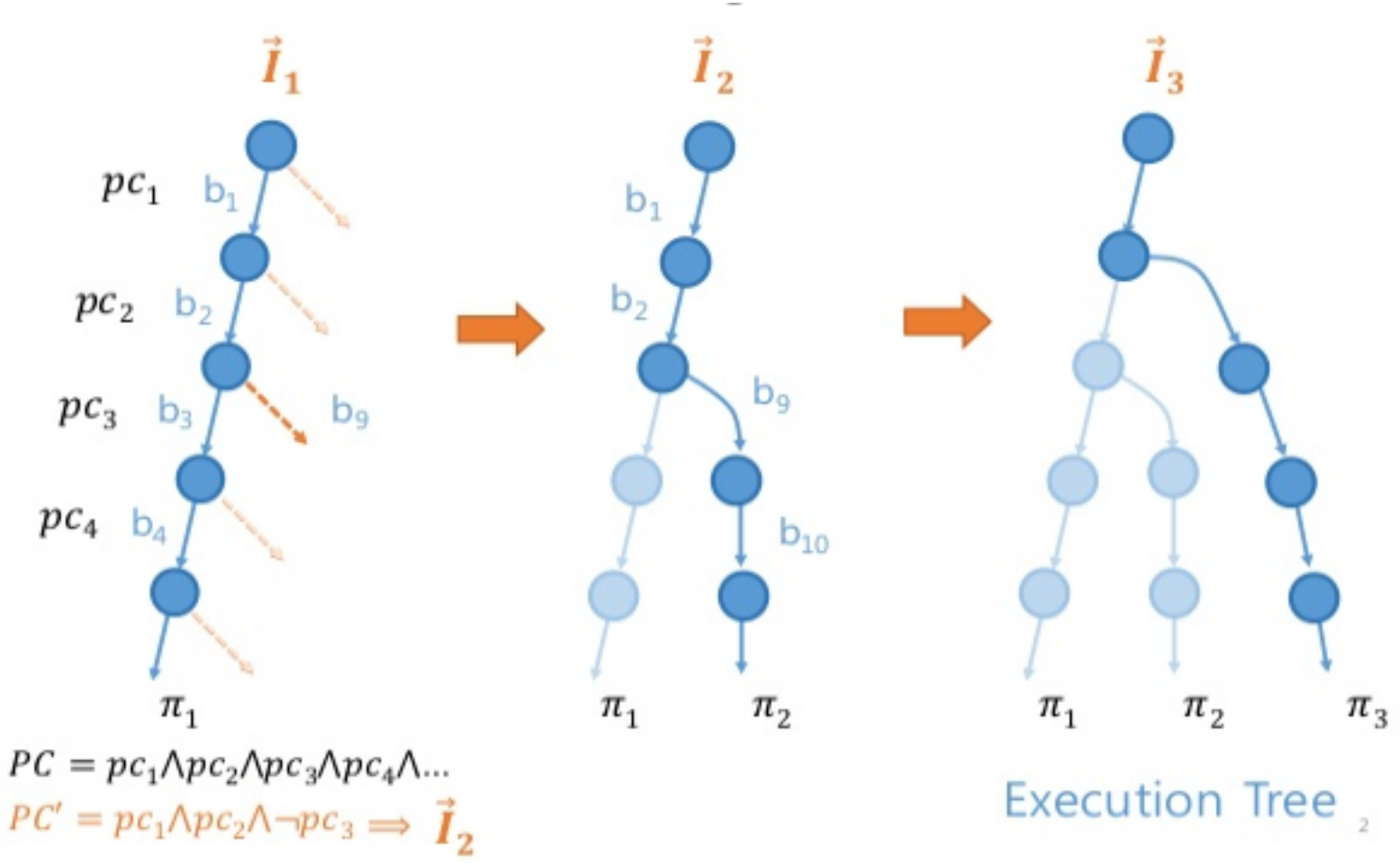
Concrete + Symbolic pytorch

利用concrete值導引探索策略，有助減少使用solver的次數

Details

- step1: 用1組 concrete input 走出一條路，同時蒐集 path constraints
- step2: 對蒐集到的 path constraints 的其中一個 constraint 作 negate，得到另外一條路的 path constraints
- step3: 用 SMT solver 解出 concrete input，重複 step1

Handling State Explosion - Concolic execution



Constraint Solving

Why optimization of constraint solving is important?

The time percentage spent on constraint solving is high.

Application	Instrs/sec	Queries	Q-size	Queries/sec		Solver(%)	
				total	STP	total	STP
[	3,914	197,282	2,868	55.1	60.0	97.8	89.8
base64	18,840	254,645	546	73.8	76.6	97.0	93.4
chmod	12,060	202,855	7,125	36.4	40.2	97.2	87.9
comm	73,064	586,485	120	189.0	201.9	88.4	82.7
csplit	10,682	244,803	2,179	49.7	52.7	98.3	92.7
dircolors	8,090	175,531	1,588	49.3	50.5	98.6	96.4
echo	227	114,830	6,852	34.8	41.7	98.8	82.3
env	21,955	379,421	664	109.1	119.8	97.2	88.5
factor	1,897	19,055	2,213	5.3	5.3	99.7	99.4
join	12,649	131,947	1,391	36.6	37.2	98.1	96.3
ln	13,420	366,926	786	103.8	115.3	97.0	87.4
mkfifo	25,331	221,308	2,144	62.3	67.4	96.6	89.3

Constraint Solving

Constraints solving poses critical performance bottleneck of symbolic execution

Many issues

- Number of constraints
- Frequent solver query
- Non-linear constraints
- black-box library call or native methods

Improving Constraint Solving Performance

The optimization of constraint solving can be roughly divided into four categories:

- Reduce the number of constraints
  - E.g., f: { $x > 10 \wedge x > 20$ } can be reduced to f: { $x > 20$ }
- Reduce the frequency of solver query
  - E.g., Reuse the constraint proof in previous query.
- Solving complicated path constraint via approximation
  - E.g., simplify constraint by concretizing: f: { $x^3+y^2 < 10.0$ } reduce to f: { $x^3+2^2 < 10.0$ } by concrete symbolic variable y with value 2.
- Maximize the potential strength of constraint solvers
  - E.g., Solve a path constraint by running several solvers in parallel.

Popular SMT Solvers

Z3

- https://github.com/Z3Prover/z3
- pip install z3-solver

CVC4

boolector

STP

...

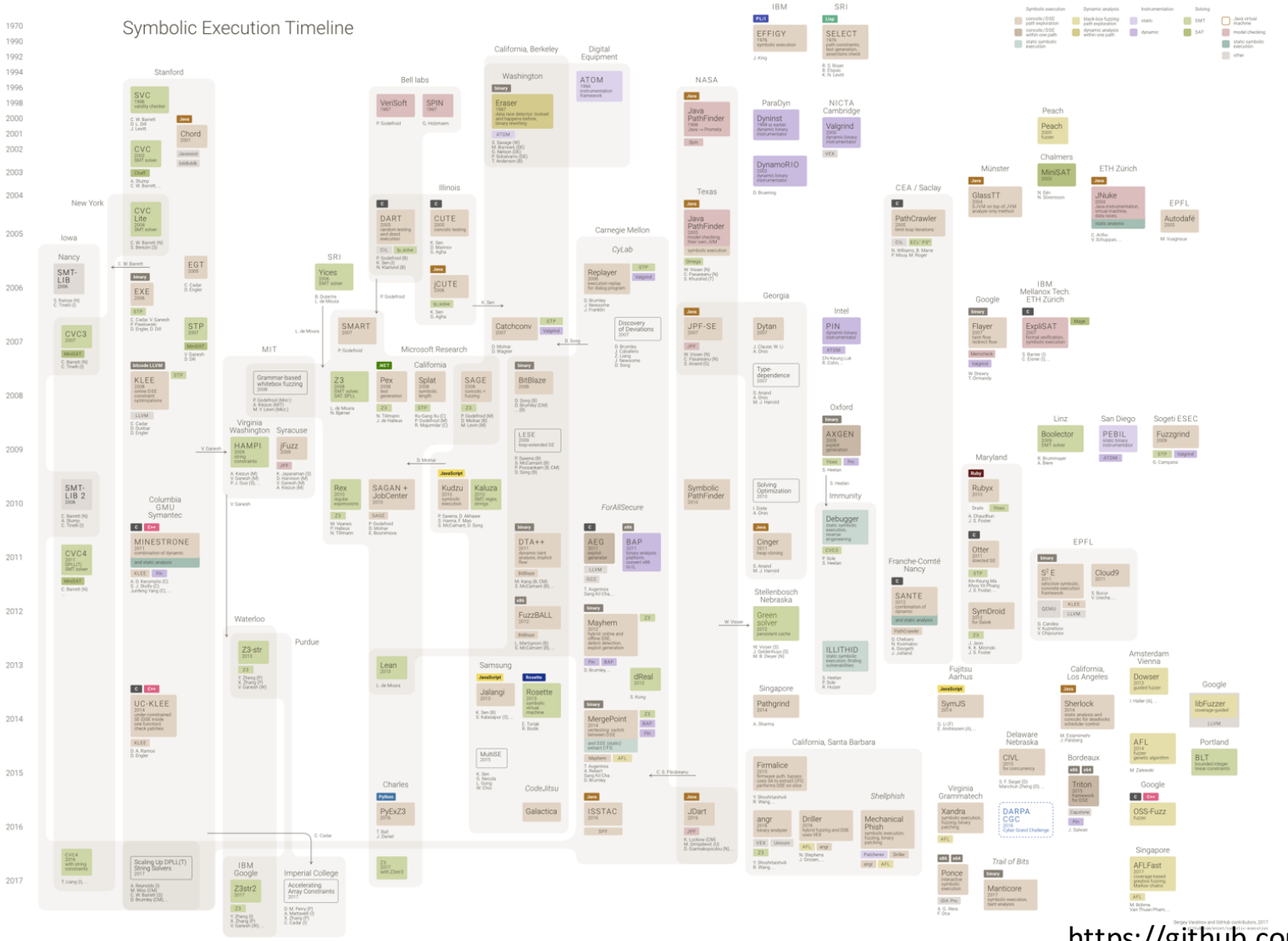
```
from z3 import *

x = Int('x')
y = Int('y')
s = Solver()
s.add(x > 2, y < 10, x + y == 7)
print s.check()  # sat
m = s.model()
print m # [y = 0, x = 7]
```

Z3 Practice: Checking Integer Overflow

```
from z3 import *
a = BitVec('a', 32)
b = BitVec('b', 32)
u_avg = UDiv(a + b, 2)
print u_avg
real_u_avg = Extract(31, 0, UDiv(ZeroExt(1, a) + ZeroExt(1, b), 2))
solve(u_avg != real_u_avg)
```

http://css.csail.mit.edu/6.858/2017/labs/lab3.html



https://github.com/enzet/symbolic-execution