CSIT 5740 Introduction to Software Security

Note set 4C

Dr. Alex LAM

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY | DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

The set of note is adopted and converted from a software security course at the Purdue University by Prof. Antonio Bianchi

Address Sanitizer (ASAN)

Address Sanitizer (ASAN)

- Programming tool useful to detect memory corruption bugs
- Initially developed by Google
- Initially used to find bugs in Chromium

Address Sanitizer (ASAN)

- ASan is programming tool useful to detect memory corruption bugs

- It is based on compile-time instrumentation (instrumentation is the act of modifying the original program to allow program memory usage checks and tracing)

Availability

- As of today ASan is available for multiple computer architecture under multiple Operating Systems:

OS	x86	x86_64	ARM	ARM64	MIPS	MIPS64	PowerPC	PowerPC64
Linux	yes	yes			yes	yes	yes	yes
OS X	yes	yes						
iOS Simulator	yes	yes						
FreeBSD	yes	yes						
Android	yes	yes	yes	yes				

- It is available in Kali too, your gcc compile should be able to access it directly (more on this later)

Address Sanitizer (ASAN)

- It is based on compile-time instrumentation
  - compile-time
    - applied when the program is compiled
    - it requires special compilation flags
    - it requires having the source code
  - Instrumentation* (this is the act of modifying the original program to allow program memory usage checks and tracing)
    - it changes how the program is compiled
    - it adds extra instructions to keep track of:
      “what the program is doing”, specifically, in terms of allocations/deallocations, memory accesses, …

Address Sanitizer (ASAN)

- Normally, to detect memory corruption, we rely on crashes
  - Crashes may not even happen in many cases, for instance:
    - the corruption affects a small amount of memory, which is not “normally” used
  - However, even bugs that normally do not crash the program, could potentially be exploited by an attacker

Address Sanitizer (ASAN)

- ASan is mostly a bug detection method more than a hardening technique * (software hardening refers to the act of applying a collection of tools, techniques, and best practices to reduce vulnerability in software).
  - In many cases, an ASan-compiled binary is still exploitable (one example at the end)

- The design idea is to use ASan during program testing to be able to detect bugs immediately, without just relaying on crashes

Sample Code

```c
#include <stdio.h>
int main() {
    unsigned long a[4];
    unsigned long index = 0;
    printf("start\n");

    index = 5;
    a[index] = 3; //buffer overflow here

    printf("end\n");

    return 0;
}
```

Sample Code compiled with ASAN

```
┌──(alex㉿kali)-[~/CSIT5740]
└─$ gcc -O0 -g -fsanitize=address -fno-omit-frame-pointer AsanExample.c -o AsanExample
```

In order to use AddressSanitizer (ASan), we need to compile and link the program using:

- "`-fsanitize=address`" switch
- add "`-O1`" or higher for a reasonable performance (ASAN will make the program slower, more on this), but this is optional for small programs
- To get nicer stack traces in error messages add "`-fno-omit-frame-pointer`"

ASAN Example

```
└─$ ./AsanExample
start
=================================================================
==66172==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7f37ad900048 at pc 0x55958b90026
55fe8660 sp 0x7fff55fe8658                                                          1 bp 0x7ffc
WRITE of size 8 at 0x7f37ad900048 thread T0
    #0 0x55958b900260 in main /home/alex/CSIT5740/lecture5/AsanExample.c:8
    #1 0x7f37af928c89 in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #2 0x7f37af928d44 in __libc_start_main_impl ../csu/libc-start.c:360
    #3 0x55958b9000d0 in _start (/home/alex/CSIT5740/lecture5/AsanExample+0x10d0) (BuildId: 1bd10b3b5
cb9ec68a47a32bc05864)                                                               fdb7819abf9

Address 0x7f37ad900048 is located in stack of thread T0 at offset 72 in frame
    #0 0x55958b9001a8 in main /home/alex/CSIT5740/lecture5/AsanExample.c:2
```

ASAN Internals

To keep track of invalid memory accesses we need:

1) **A data structure to keep track of valid memory regions, i.e., memory regions where program's variables are legitimately stored, these are the memory regions that have been allocated to the variables** (called *"addressable"* in the original paper)
   - Shadow memory → data structure to keep track of memory usage
   - A data structure to keep track of which memory regions are "addressable" (i.e., used by program's variables) and which memory regions are not addressable

2) **Update such a data structure every time the program creates/destroys a program's variables**
   - Shadow memory updating → update tracked memory

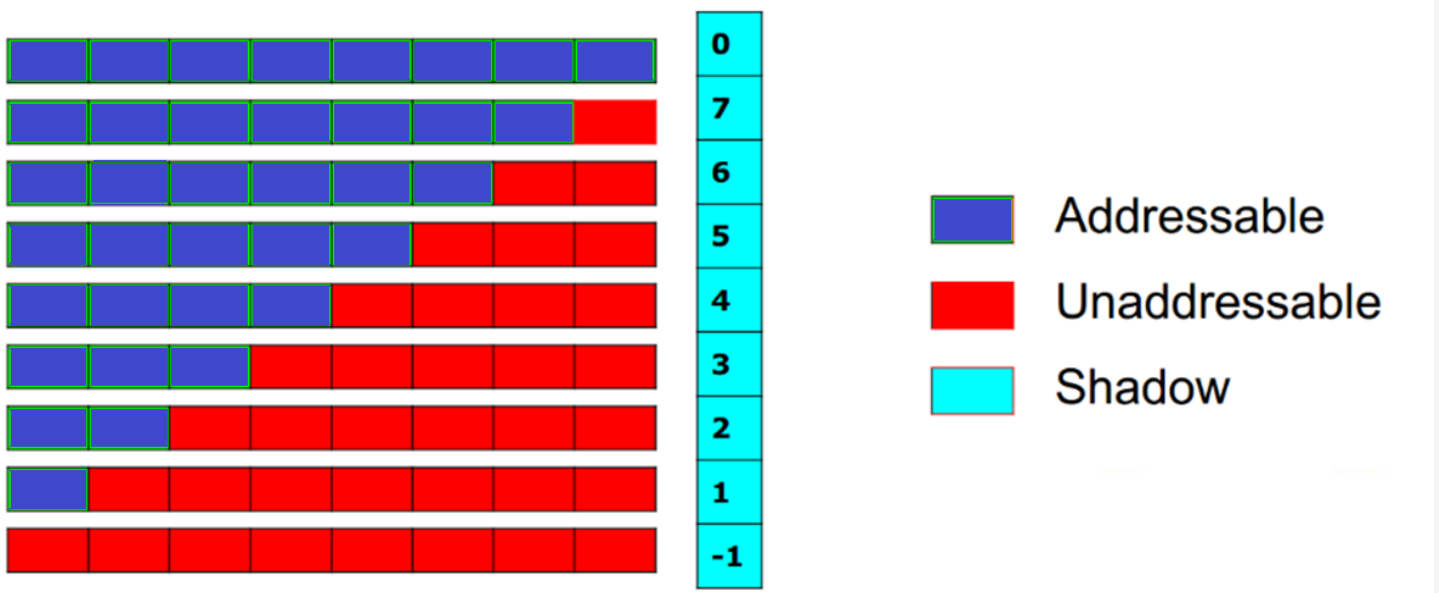
3) **Check if every memory access is inside a valid memory region**
   - Memory access instrumentation → check if memory accesses are only in addressable regions

ASAN Internals: addressability

- ASan considers the memory being allocated in the program to be addressable (i.e. memory allocated to the variables, arrays, etc), these are the memory units that could be accessed by the program
- Other part of the memory not yet allocated is considered *unaddressable (poisoned)*
- A program is only supposed to access the allocated/addressable memory only
  - For example for the array `int array[8]` defined in the program, since it is defined, it is allocated memory, all the element `array[i], i=0,…,7` are addressable but not `array[-1]` nor `array[8]`
- When a program tries to access unaddressable *(poisoned)* memory, a buffer overflow is happening, ASan will stop the program
- ASan also protects the stack/heap by setting up “redzones” before the stack/heap and after the stack/heap. If a program tries to access a “redzone”, it is likely to be a stack/heap overflow access and ASan will stop the program

ASAN Internals: Shadow Memory

- For every 8 bytes of allocated memory ASan uses 1 byte of **shadow memory** to keep track of the usage information of the allocated memory. Shadow memory is the technical term used by Google to describe the memory that is use for storing administration information regarding the allocated memory
- The content of the **shadow memory** depends on state (specifically, how many bytes are used) within the aligned 8-byte word



ASAN Internals: Shadow Memory

- Apart from the shadow memory values on the previous slide, there are a lot more other values in the real ASan implementation. You only need to know about the ones enclosed by the blue rectangles.

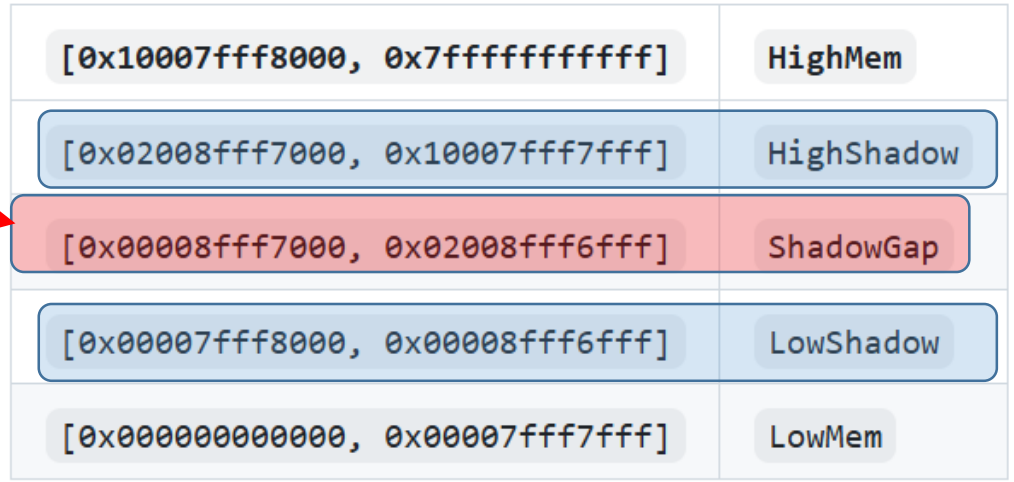
```
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
```

ASAN Internals: Shadow Memory

- Shadow memory is located in a dedicated memory area, in 64-bit systems, the shadow memory for storing administration information of a particular memory is calculated as :

```
Shadow_address  = (Mem >> 3) + 0x7fff8000
```

Where **Shadow_address** is the address of the shadow memory, **Mem** is the memory that is being "looked after", "**>> 3**" is the "shift logical right by 3 bits" operation

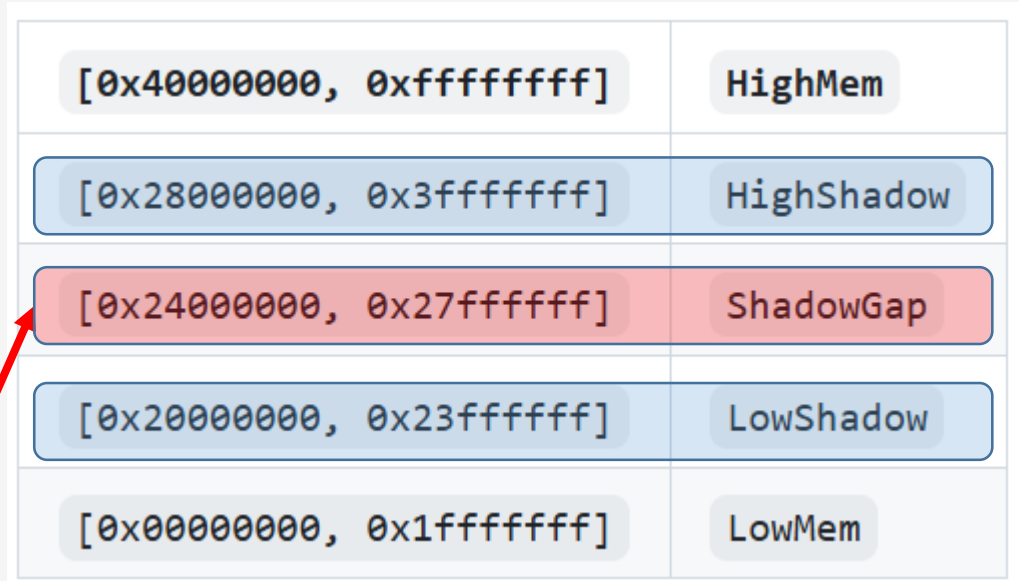


The ShadowGap region is unaddressable. If a program tries to directly access a memory location in the shadow region, it will crash.

ASAN Internals: Shadow Memory

- Shadow memory is located in a dedicated memory area, in 32-bit systems, the shadow memory for storing administration information of a particular memory is calculated as :

```
Shadow_address = (Mem >> 3) + 0x20000000
```



The ShadowGap region is unaddressable. If a program tries to directly access a memory location in the shadow region, it will crash.

ASAN Internals: Shadow Memory

- **Example (assume 32-bit system) :**

  if 8 bytes at 0x40000010 are addressable

  → 1 byte at (0x40000010>>3=0x08000002) + (0x20000000) = 0x28000002 contains **0x00**

  Other bytes are “poisoned”, i.e., flagged as non allocated (0xf9 below indicates that they are “global red zones” indicate the unaddressable memory in the memory region that stores global variables)

  0x28000000: [0xf9 0xf9 **0x00** 0xf9 0xf9 0xf9 0xf9 0xf9]

ASAN Internals: Shadow Memory

- **Example (assume 32-bit system) :**

  if 4 bytes at 0x40000098 are addressable

  → 1 byte at (0x40000098>>3=0x08000013) + (0x20000000) = 0x28000013 contains **0x04**

  Other bytes are flagged as non allocated (again 0xf9 below indicates that they are “global red zones”)

  0x28000010: [0xf9 0xf9 0xf9 **0x04** 0xf9 0xf9 0xf9 0xf9]

ASAN Internals

2) Shadow memory filling
   - Shadow memory is updated
     - when the program starts, to keep track of global variables
     - when heap memory is allocated (malloc)
     - when a new stack frame is created (a function is called) and destroyed (a function returns)
   - Red-Zones: around allocated stack variables or heap regions, ASAN adds areas of poisoned memory (in the shadow memory)
     - it allows overflow detection

ASAN Internals

3) Memory Access Instrumentation

- for 8 byte accesses

```
ShadowAddr = (Addr >> 3) + Offset;
if (*ShadowAddr != 0){
    ReportAndCrash(Addr);
}else{
    //…
}
```

- for N byte accesses (N<8)

```
ShadowAddr = (Addr >> 3) + Offset;
 if ((*ShadowAddr != 0) && (*ShadowAddr-1 < ((Addr&7)+N-1))){
    ReportAndCrash(Addr);
}else{
        //...
}
```

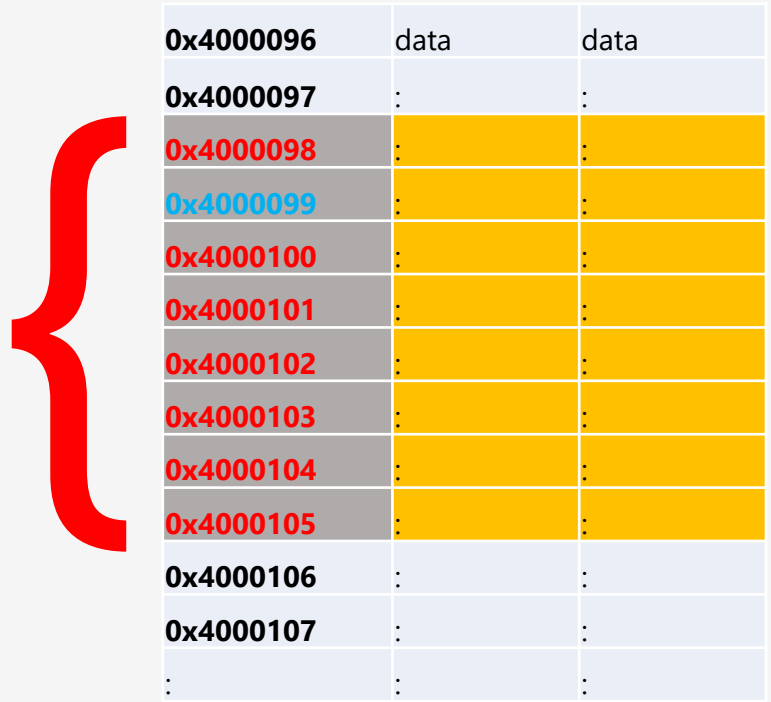
Memory blocks

- Given a memory address, if we divide memory into 8-byte blocks. To find the block number of an arbitrary memory address "`Mem`", we just divide the memory address by 8 and convert it to integer (i.e. `(int)(Mem/8)`)
- This is equivalent to **`Mem >> 3`**
- For example if Mem is 0x4000099:
  - 0x4000099 = 0100 0000 0000 0000 0000 1001 1001

=> 0x4000099 >>3 = 1000 0000 0000 0000 0001 0011

= 0x800013

This is block 0x800013



ASAN Internals: Memory Access Instrumentation

- **Example**
  check to see if accessing 4 bytes at 0x40000098 is allowed
- Let's take a look at the shadow memory for storing the status of 0x40000098
- Shadow memory address for the data at 0x40000098 is:

ShadowAddr = (0x40000098 >> 3) + Offset; //Offset=0x20000000,

= 0100 0000 0000 0000 0000 0000 1001 1000 >>3 + 0x20000000

= 1000 0000 0000 0000 0000 0001 0011 + 0x20000000

= 0x28000013

0x28000010: [0xf9 0xf9 0xf9 **0x04** 0xf9 0xf9 0xf9 0xf9]

ASAN Internals: Memory Access Instrumentation

- **Example**
  check to see if access 4 bytes at 0x40000098 is allowed

int* X = … //4-byte allocation at 0x40000098

*X = 123; // run the instrumented code

// *X = 123 becomes:

```
ShadowAddr = (Addr >> 3) + Offset;
 if ((*ShadowAddr != 0) && (*ShadowAddr-1 < ((Addr&7)+N-1))){
    ReportAndCrash(Addr);
}else{
       //...
}
```

this part checks if the whole 8-byte memory block is addressable/allocated. If this is the case, any data access to any address of this block will not overflow the block and it will return False. Otherwise if not all the 8 bytes are allocated, it will return True.

this part checks if the memory access overflows the allocated amount in the 8-byte memory block

0x28000010: [0xf9 0xf9 0xf9 **0x04** 0xf9 0xf9 0xf9 0xf9]

ASAN Internals: Memory Access Instrumentation

- **Example**
  check to see if access 4 bytes at 0x40000098 is allowed

```
ShadowAddr = (Addr >> 3) + Offset;
 if ((*ShadowAddr != 0) && (*ShadowAddr-1 < ((Addr&7)+N-1))){
    ReportAndCrash(Addr);
}else{
       //...
}
```

ShadowAddr = 0x28000013

- *ShadowAddr = 0x4 → *ShadowAddr !=0 → **True** **(False means not all the 8 bytes are addressable/allocated)**

- ((Addr&7)+N-1) = 0100 0000 0000 0000 0000 0000 1001 1000 & 0…0111 + 0x3 = 0x3

→ 0x4 <= 0x3 → **False** **(this part checks if the memory access overflows the allocated amount)**

→ **True** && **False** = **False** → the memory access is ok

0x28000010: [0xf9 0xf9 0xf9 **0x04** 0xf9 0xf9 0xf9 0xf9]

ASAN Usage

- ASAN can be bypassed by an attacker, for instance:
  - “Precise” writes

```
unsigned int a1[10]; //at 0x556c7869b280
unsigned int a2[10];
unsigned int is_admin=0; //at 0x556c7869b340
int main() {
    unsigned long index =48 //exact offset for accessing is_admin from a1[]
    printf("start\n");
    printf("%p %p\n", a1, &is_admin);
    a1[index] = 1;
    printf("%d\n", is_admin);
    printf("end\n");
    return 0;
}
```

ASAN Usage

- ASAN can be bypassed by an attacker
- In fact, its primary goal is to be able to identify non-crashing memory corruption, during:
  - manual testing
  - automatic testing
    - test suite
    - fuzzing (more on this later)

ASAN Usage

- ASAN checks are added after other compiler optimization passes
  → the compiler may remove some checks if not useful

- It introduces a slow-down of about 2 times
- It increases memory consumption

ASAN References

- Paper
  - https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf
- Slides
  - https://www.usenix.org/sites/default/files/conference/protected-files/serebryany_atc12_slides.pdf

Automated Analysis Introduction

Automated Analysis

- Up to now, all our security analysis has been manual
  - Run
  - Decompile
  - Study the code
  - Develop an exploit
  - ...
- Obviously, this approach has scalability issues, and may not be sustainable
- A big chunk of modern security research focuses on automate bug finding (and even bug exploitation!)

Dynamic vs. Static Analysis

- Intuitively we could say that
  - methods that require executing the code are “dynamic”
  - methods that do not require executing the code are “static”

- Analysis in GDB or strace → Dynamic
- Decompilation and Reversing the code back to C program → Static

Automated Analysis

- We use an automated approach to figure out some properties of the code
  - does this code contain a memory corruption bug?
  - does this code contain any vulnerability?
  - does this code contain any unwanted behavior (malware analysis)?

Automated Analysis

- Dynamic Analysis
- Symbolic Execution
  - Depending on how it is performed it can be considered either a dynamic or a static technique
- Static Analysis

Automated Dynamic Analysis

- Dynamic Analysis: Fuzzing

- The basic idea:
  1) Generate inputs for the program (e.g., strings to stdin)
  2) Run the program with those inputs
  3) Detect if the program reaches a certain state (e.g., it crashes)

Fuzzing: Example

- **How do I trigger the condition: bug() is called**

```
x = int(input())
if x >= 10:
        if x < 100:
                bug()
        else:
                print "You lose!"
else:
        print "You lose!"
```

Fuzzing: Example

- **How do I trigger the condition: bug() is called**

```
x = int(input())
if x >= 10:
        if x < 100:
                bug()
        else:
                print "You lose!"
else:
        print "You lose!"
```

- Try "1" → *"You lose!"*
- Try "2" → *"You lose!"*
- …
- Try "10" → bug()

Symbolic Execution

- Symbolic Execution

- The basic idea:
  1) Replace inputs with symbolic values
  2) Explore all possible paths, collecting constraints
  3) Use a constraint solver to compute how to reach a specific code location

Symbolic Execution

- **How do I trigger the condition: bug() is called**

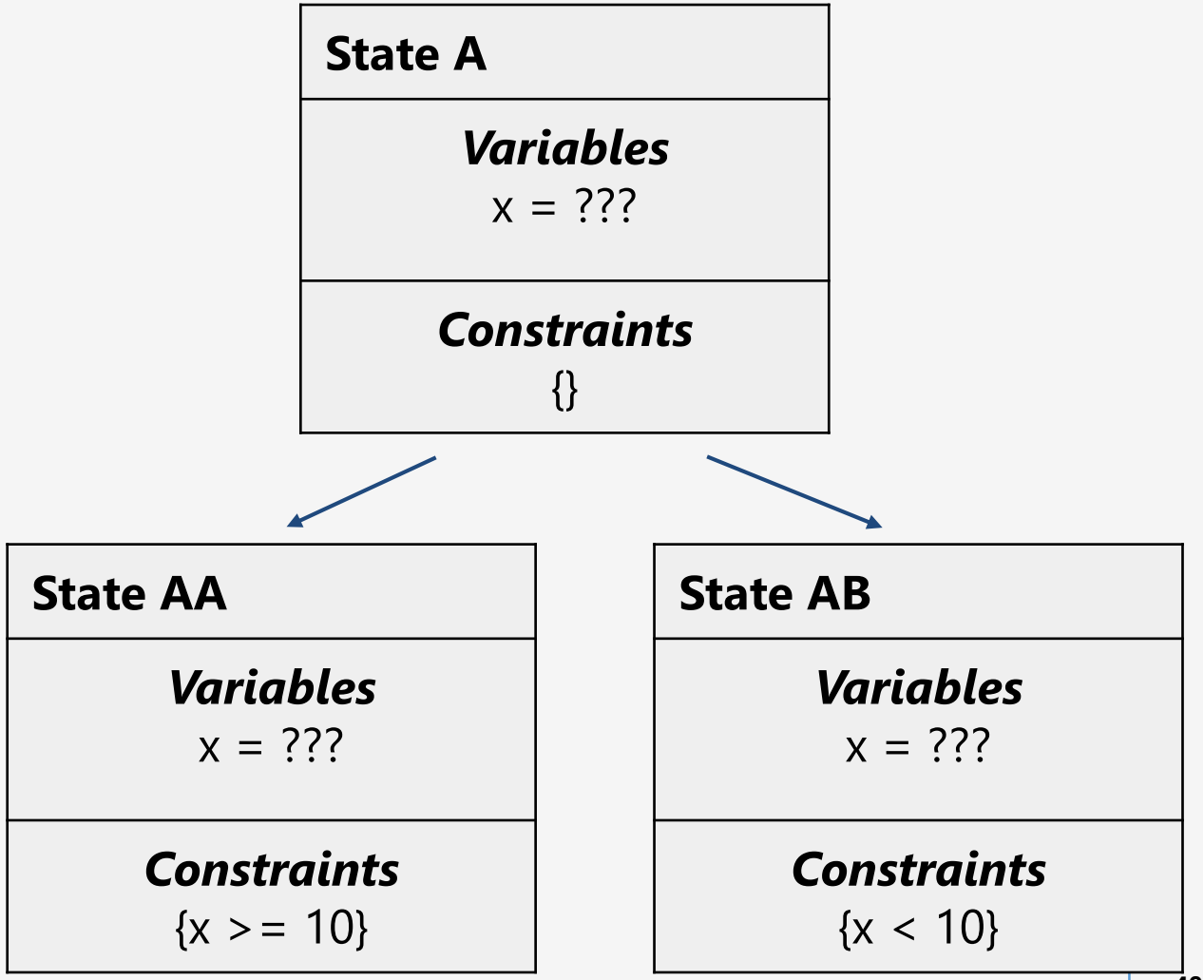
```
x = int(input())
if x >= 10:
        if x < 100:
                bug()
        else:
                print "You lose!"
else:
        print "You lose!"
```

State A
Variables x = ???
Constraints {}

Symbolic Execution

- **How do I trigger the condition: bug() is called**

```
x = int(input())
if x >= 10:
        if x < 100:
                bug()
        else:
                print "You lose!"
else:
        print "You lose!"
```



Symbolic Execution

- **How do I trigger the condition: bug() is called**

```
x = int(input())
if x >= 10:
        if x < 100:
                bug()
        else:
                print "You lose!"
else:
        print "You lose!"
```

State AA
Variables x = ???
Constraints {x >= 10}

State AB
Variables x = ???
Constraints {x < 10}

Symbolic Execution

- **How do I trigger the condition: bug() is called**

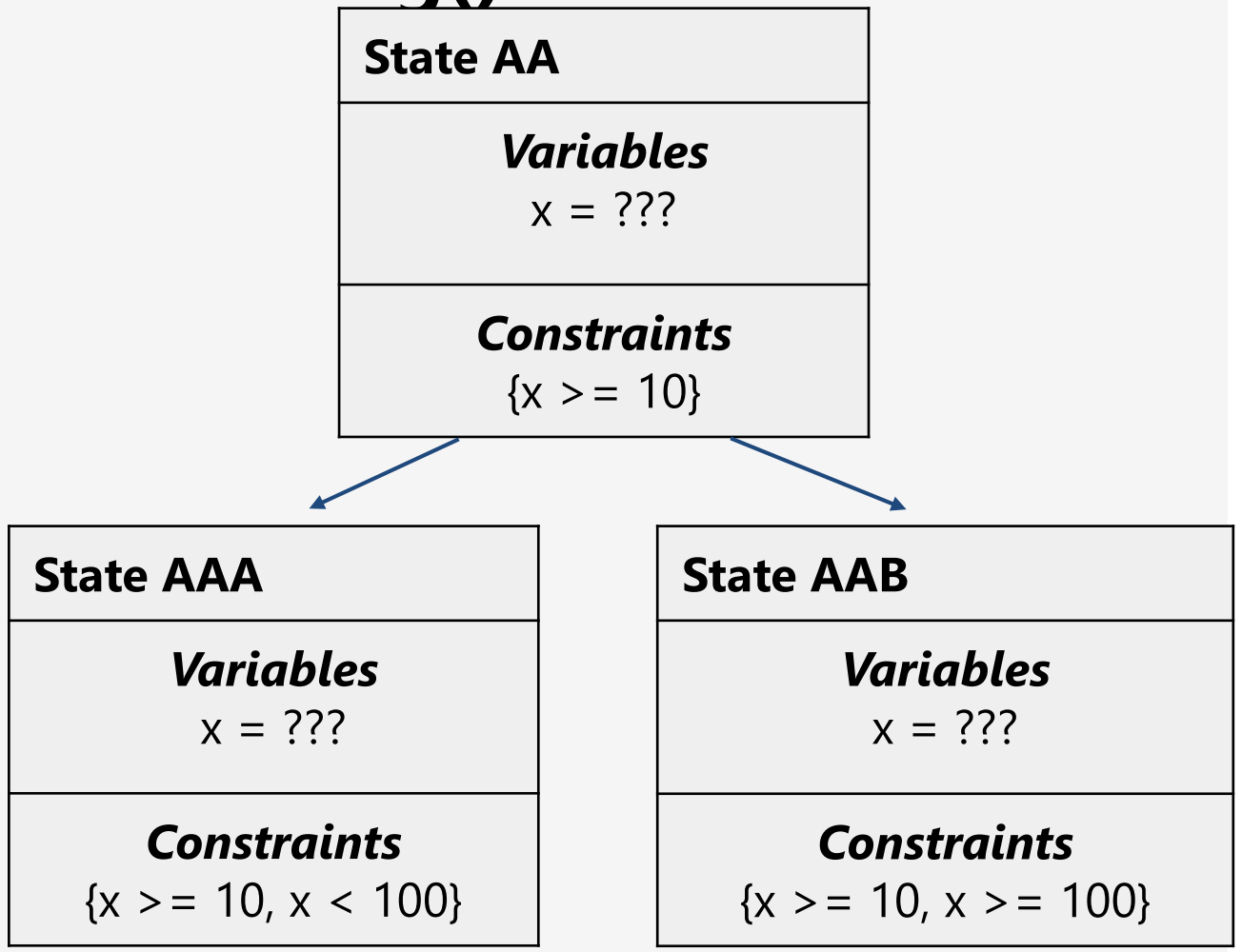
```
x = int(input())
if x >= 10:
        if x < 100:
                bug()
        else:
                print "You lose!"
else:
        print "You lose!"
```

State AA
Variables x = ???
Constraints {x >= 10}

Symbolic Execution

- **How do I trigger the condition: bug() is called**

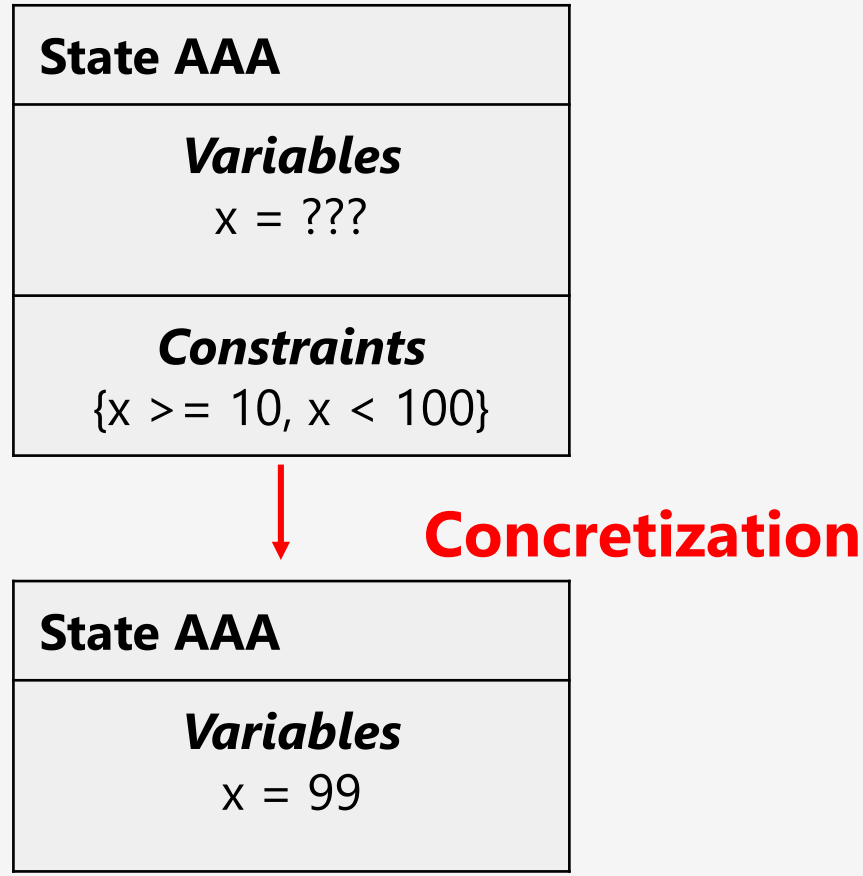
```
x = int(input())
if x >= 10:
        if x < 100:
                bug()
        else:
                print "You lose!"
else:
        print "You lose!"
```



Symbolic Execution

- **How do I trigger the condition: bug() is called**

```
x = int(input())
if x >= 10:
        if x < 100:
                bug()
        else:
                print "You lose!"
else:
        print "You lose!"
```



Automated Static Analysis

- Analyze code without running it
- Many different static analyses exist

- A very basic example:
  - Disassemble the code and then grep for calls to specific functions
    - It could be useful to detect if particularly “dangerous” functions can be used (e.g., strcpy, gets, ...)

Automated Static Analysis

- A more complex case: static taint analysis
  - Useful to determine how information “flows” within a program
  - For instance: is the first argument of **printf** coming from user’s input?

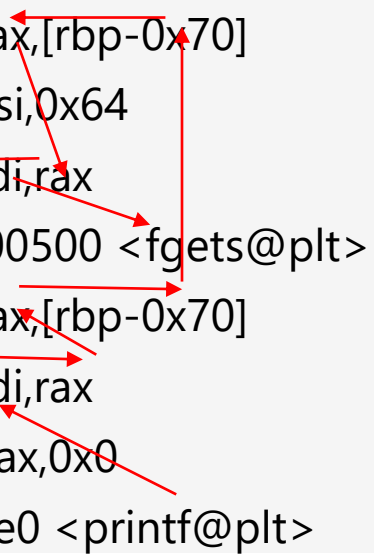
- Is the first argument of printf coming from user's input?

```
void main(){
    char a[100+1];
    fgets(a, 100, stdin);
    printf(a);
}
```

The first argument of printf is stored in rdi (at 400649).
We follow backward how rdi at 400649 is set, and we understand that it points to the same location pointed by rdi at 40063d, which is used as the first argument of fgets.
Therefore the first argument of printf can be controlled by the user → format string vulnerability

Let's check how the first argument of printf is set

```
400634:   48 8d 45 90          lea     rax,[rbp-0x70]
400638:   be 64 00 00 00       mov     esi,0x64
40063d:   48 89 c7             mov     rdi,rax
400640:   e8 bb fe ff ff       call    400500 <fgets@plt>
400645:   48 8d 45 90          lea     rax,[rbp-0x70]
400649:   48 89 c7             mov     rdi,rax
40064c:   b8 00 00 00 00       mov     eax,0x0
400651:   e8 8a fe ff ff       call    4004e0 <printf@plt>
```



Automated Static Analysis

- Another example: **Control-flow Graph**
  - We can represent each function's code as a graph in which every node is a basic block
    - a list of instructions that are always executed sequentially
    - alternative definition: a straight-line piece of code without any jumps
    - Depending on the definition "calls" may or may not split blocks
  - Every edge is a possible transitions between these blocks
  - If we extend these graph to the entire program, it is called Inter-procedural Control Flow Graphs, or iCFG

Automated Analysis: Error Types

- FP: False Positive
  - The analysis says "X can happen" while, in fact, it cannot happen
- FN: False Negative
  - The analysis says "X will never happen", while, in fact, it can happen

Fuzzing Issues

- Code Coverage
  - Only a very limited subset of paths are covered
    - Depending on the scenario: false negatives

```
x = int(input())
if x >= 10:
        if x == 0x424242424242:
                bug()
        else:
                print "You lose!"
else:
        print "You lose!"
```

Symbolic Execution Issues

- In theory, it can explore all possible code paths.
  However, it has severe scalability issues.

- Path Explosion
  - Intrinsically symbolic execution is an exponential problem

- Constraint Complexity
  - The collected constraints may easily become too complex to be solvable

Static Analysis Issues

- In theory, it can reason about the entire code.
  However, it has to choose tradeoffs between scalability and precision

- Many static analysis are undecidable
  - For instance, pointer aliasing
    - Deciding if two pointers may point to the same memory location

- Possible solution: over-approximation
  - For instance, in building a callgraph: design an analysis which returns all possible edges, but potentially also some extra edge

Evasion and Obfuscation

- Evasion → Malicious code tries to avoid being detected
  - Example: detect if "under analysis”
    - check if a debugger (GDB) is attached
    - check the execution environment to detect if it is an analysis environment

- Obfuscation → Make the code hard to analyze
  - Generate code at runtime
    - Download malicious code
  - Use non-standard compilers or obfuscating compiler passes
  - …

Real-world program analysis applications

- In many cases: combination of:
  - static analysis + dynamic analysis + symbolic execution

- For instance:
  - Static Analysis can identify functions that may contain vulnerabilities (e.g., they call potentially unsafe functions such as strcpy)
  - Dynamic Analysis (Fuzzing) can be used to find program's inputs reaching those interesting functions
  - Symbolic Execution can be used to bypass "hard-to-bypass" checks