CS 578: CYBER-SECURITY
PART IV: MALWARE

Sanghyun Hong
sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab

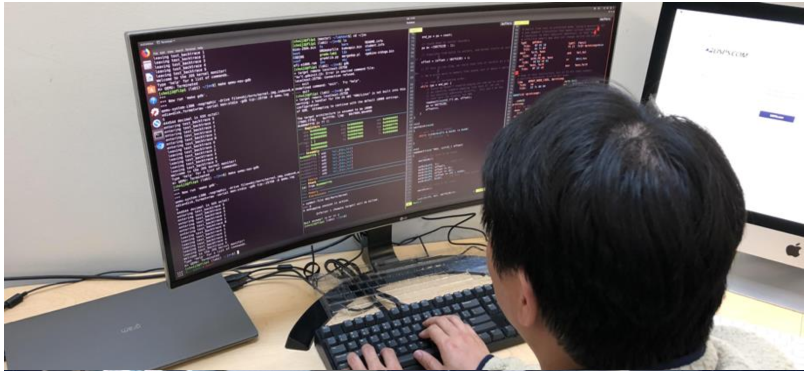
PRELIMINARIES

HUMANS MAKE ERRORS

- Suppose that we manufacture products
- We make errors if
  - We are under stress
  - We worked too many hours
  - We face a quick production cycle (e.g., one day)
  - … (many more)

HUMANS MAKE ERRORS

- We develop software
  - Humans are prone to making errors
  - Humans make more mistakes if
    - They are too stressful from work
    - They are too stressful from life
    - Work is hard
    - Worked too much hours (160+ hrs/wk)
    - A quick development cycle (sprints)
    - … (many more)



Modern software is complex

- Google Chrome
  - +4M lines of pure code in 10 yrs ago

Shashwat Anand
Participated in Google Summer of Code. · Author has **100** answers and **465.9K** answer views · Updated 10y

4,490,488 lines of code, 5,448,668 lines with comments included, spread over 21,367 unique files.

Used Cloc [http://cloc.sourceforge.net/] just like Dan Loewenherz did for the question How many lines of code are in the Linux kernel?

MODERN SOFTWARE IS COMPLEX

- Google Chrome
  - +4M lines of pure code in 10 yrs ago

Shashwat Anand
Participated in Google Summer of Code. · Author has **100** answers and **465.9K** answer views · Updated 10y

4,490,488 lines of code, 5,448,668 lines with comments included, spread over 21,367 unique files.

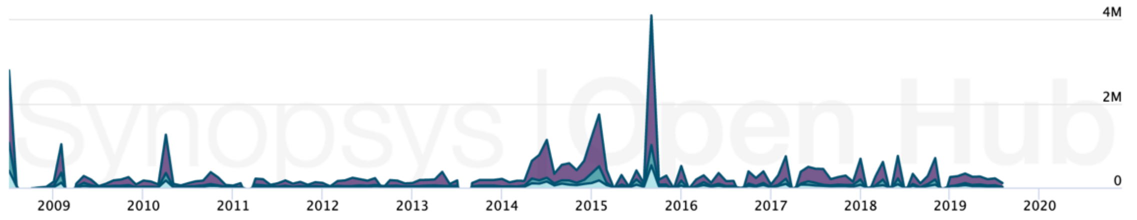
Used Cloc [http://cloc.sourcefo
How many lines of code are in th

Total Lines :	34,900,821	Code Lines :	25,683,389	Percent Code Lines :	73.6%
Number of Languages :	36	Total Comment Lines :	4,603,400	Percent Comment Lines :	13.2%
		Total Blank Lines :	4,614,032	Percent Blank Lines :	13.2%

  - >34M lines these days..

Code, Comments and Blank Lines

Zoom 1yr 3yr 5yr 10yr **All**



MODERN SOFTWARE IS COMPLEX

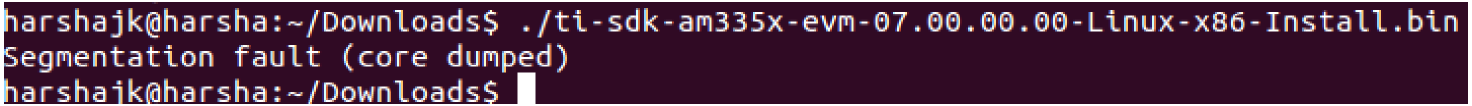
- Others
- Linux kernel
  - >12M lines of code in 2015
  - >27M lines of code in 2020

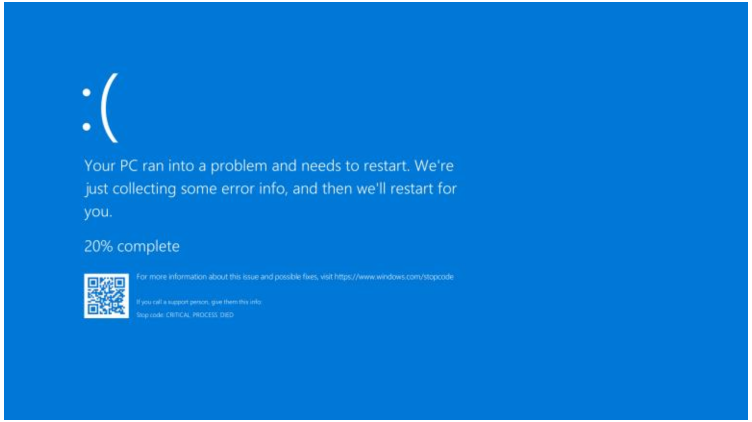
- Android
  - Android 1.6: >4.5M lines in 2009
  - Android 5.1: > 9M lines in 2014
  - Android 8.0: > 25M lines in 2017

- Humans are prone to making errors
- Work environment often makes people to more prone to making errors in code
- The complexity in software makes it more difficult for humans to follow the code (Complexity: $O(N^2)$ where N = lines of code)
- …

WHAT CAN GO WRONG IF WE MAKE MISTAKES IN OUR SOFTWARE?

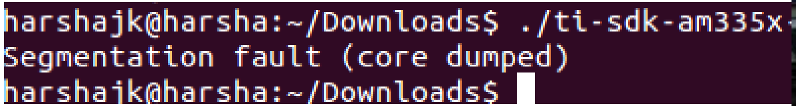
- Crash

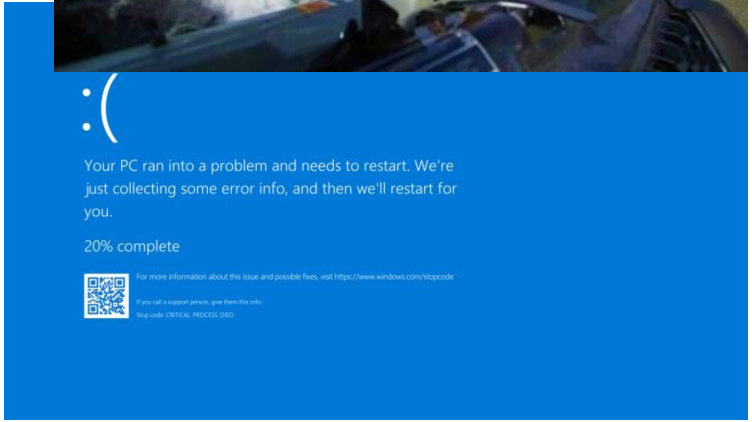




WHAT CAN GO WRONG IF WE MAKE MISTAKES IN OUR SOFTWARE?

- Crash

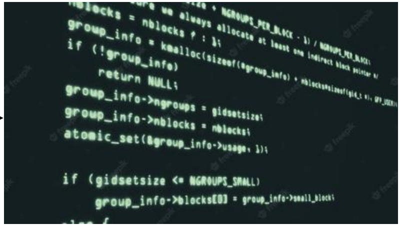




WHAT CAN GO WRONG IF WE MAKE MISTAKES IN OUR SOFTWARE?

- Crash
- A hack

1. Identify our mistakes in code



2. Build an exploit of the mistakes
   (control the error to do sth. else)

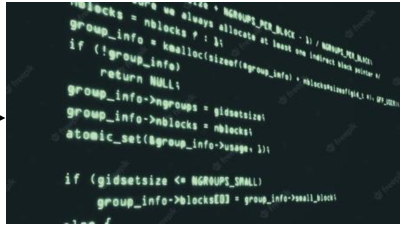
3. Do malicious things
   (e.g., get an admin access of systems)

WHAT CAN GO WRONG IF WE MAKE MISTAKES IN OUR SOFTWARE?

- Crash
- A hack

1. Identify our mistakes in code

2. Build an exploit of the mistakes
   (control the error to do sth. else)



3. Do malicious things
   (e.g., get an admin access)

WHAT CAN GO WRONG IF WE MAKE MISTAKES IN OUR SOFTWARE?

- Vulnerability
  - An error (= a bug) that can be exploited by an adversary
  - The attacker can alter the intended operation of a software in a malicious way
  - You can find vulnerabilities in, e.g., CVE database or CWE database

- Exploit
  - An input that triggers a vulnerability with malicious intent
  - A proof-of-concept program that demonstrates how an adversary is likely to use the vuln.
  - You can find sample exploits from, e.g., Metasploit

Motivating example: goto fail – cont'd

- In 2014

Anatomy of a "goto fail" – Apple's SSL bug explained, plus an unofficial patch for OS X!

MOTIVATING EXAMPLE: GOTO FAIL – CONT'D

- In 2014

About the security content of iOS 7.0.6

This document describes the security content of iOS 7.0.6.

iOS 7.0.6

- **Data Security**

  Available for: iPhone 4 and later, iPod touch (5th generation), iPad 2 and later

  Impact: An attacker with a privileged network position may capture or modify data in sessions protected by SSL/TLS

  Description: Secure Transport failed to validate the authenticity of the connection. This issue was addressed by restoring missing validation steps.

  CVE-ID

  CVE-2014-1266

Why???

What was the mistake??

MOTIVATING EXAMPLE: GOTO FAIL – CONT'D

- Error checking code
  - If there are 'errors' in 'err'
  - The code moves to 'fail';

- The code in the red square is okay
  - They run SHA1 and check errors

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
```

MOTIVATING EXAMPLE: GOTO FAIL – CONT'D

- Error checking code
  - If there are 'errors' in 'err'
  - The code moves to 'fail';

- The code in the red square is okay
  - They run SHA1 and check errors

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
```

MOTIVATING EXAMPLE: GOTO FAIL – CONT'D

- Error checking code
  - If there are 'errors' in 'err'
  - The code moves to 'fail';

- The code above the red square is okay
  - They run SHA1 and check errors

- The code in the red boxes:
  - It does not fall into any if statement
  - It always leads to "goto fail;"
  - It makes us skip the verification step

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
```

MOTIVATING EXAMPLE: GOTO FAIL – CONT'D

- How to exploit this mistake?
  - Suppose an attacker runs public Wi-Fi
  - You can create 'PDX Free WiFi' / 'Google Starbucks WiFi' / 'eduroam' / …
  - The attacker sends a crafted TLS packet
  - Make you choose SHA1
  - Trigger the "goto fail;"
  - Force your browser to choose weak algo.

Best public cryptanalysis

12-round RC5 (with 64-bit blocks) is susceptible to a differential attack using 2^{44} chosen plaintexts.[1]

  - Now the attacker can see all your comm.

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
```

MOTIVATING EXAMPLE: GOTO FAIL

- Small mistake; big impact
  - A mistake: adds one additional line of ‘goto fail’
  - Result : attackers may hijack a TLS protected connection
  - Impact : attackers may read/modify all TLS connections from iOS/MacOS
- Implications
  - Even a simple mistake could lead to a disaster
  - Errors are not arbitrarily happening; not like natural disaster
  - Errors can be controlled (‘exploited’) by attackers

HOW TO IDENTIFY ERRORS (= VULNERABILITIES)?

PROGRAM ANALYSIS

- Static analysis
  - Examine source code without running them
  - Search for the patterns that lead to security vulnerabilities – signatures
  - Use these signatures on the source code-level

- Dynamic analysis
  - Examine source code + running the code to observe program behaviors
  - Search for the (runtime) patterns that lead to security vulnerabilities – signatures
  - Use these signatures at runtime – help identify vulnerabilities that only manifest at runtime

PROGRAM ANALYSIS

- Static analysis
  - Examine source code without running them
  - Search for the patterns that lead to security vulnerabilities – signatures
  - Use these signatures on the source code-level

- Implications
  - At that time, most vulnerability discovery depends on manual analyses
  - There could be unresolved or unknown vulnerabilities in the source code
  - Those vulnerabilities could be used for 0-day attacks (or miscreants sell them)

PROGRAM ANALYSIS – AVG: MOSTLY STATIC

- Potential solutions: automatic vulnerability generation
  - Software patch : patch the source code (or the program) with vulnerabilities
  - Data patch : patch the data input that may trigger the vulnerability
  - Input filter : remove or rewrite potentially malicious inputs, e.g., device I/Os
  - Signature : identify and remove malicious patterns from the code or inputs!

PROGRAM ANALYSIS – AVG: MOSTLY STATIC

- Problems in manual analysis
  - Manual signature generation is slow
  - It may employ heuristics that fail in many real-world settings
  - Unknown or unresolved vulnerabilities can be exploited much faster by adversaries
  - Oftentimes, the analysis is done by looking at exploits – not an ideal practice

PROGRAM ANALYSIS – AVG: MOSTLY STATIC

- Brumley *et al.*
  - Propose a new class of signatures
  - Not specific to details, e.g., whether the signature can hijack the control of a program
  - But specific to, e.g., whether an input (potentially) results in unsafe execution

- Overview
  - Vulnerability signature:
    - An input x - executing x will result in unsafe execution
  - Execution trace: T(P, x) – run program P on input x
  - Vulnerability condition: a condition for the vulnerability, e.g., heap overflow
    - Representation : how to express a vulnerability as a signature
    - Coverage : measured by a false-positive rate
  - $L_{p,c}$: consists of a set of all inputs x that satisfy the vulnerability condition

PROGRAM ANALYSIS – AVG: MOSTLY STATIC

- An example
  - P: the program on the left
  - x: g/AAAA
  - T: {1, 2, 3, 4, 6, 7, 8, 9, 8, 10, 11, 10, 11 10, 11, 10, 11, 10, 11}
  - c: heap overflow

```
1  char *get_url(char inp[10]){
2    char *url = malloc(4);
3    int c = 0;
4    if(inp[c] != 'g' && inp[c] != 'G')
5      return NULL;
6    inp[c] = 'G';
7    c++;
8    while(inp[c] == ' ')
9      c++;
10   while(inp[c] != ' '){
11     *url = inp[c]; c++; url++;
12   }
13   printf(``%s'', url);
14   return url;
15 }
```

PROGRAM ANALYSIS – AVG: MOSTLY STATIC

- Specifics
  - Vulnerability signature:
    - An input x - executing x will result in unsafe execution
    - *MATCH(x)* -> *EXPLOIT* or *BENIGN*
      - $x \in L_{p,c}$: then *MATCH(x)* = *EXPLOIT*
      - $x \notin L_{p,c}$: then *MATCH(x)* = *BENIGN*

  - Vulnerability condition:
    - $c = \Gamma \times D \times M \times K \times I \rightarrow$ {EXPLOIT, BENIGN}
    - Γ is the memory state
    - D is the set of variables defined
    - M is the program's map from memory to values
    - K is the continuation stack
    - I is the next instruction to execute

PROGRAM ANALYSIS – AVG: MOSTLY STATIC

- Specifics
  - Vulnerability representations
    - Turing machine signatures : precise, yet may not terminate
    - Symbolic constraint signatures : approximate looping, always terminate
    - Regular expression signatures : approximate elementary constructs, efficient
    - Please refer to the paper for two points:
      - What does it mean by these signatures, what are the definitions and use cases?
      - What do we expect about the utility-efficiency trade-offs?

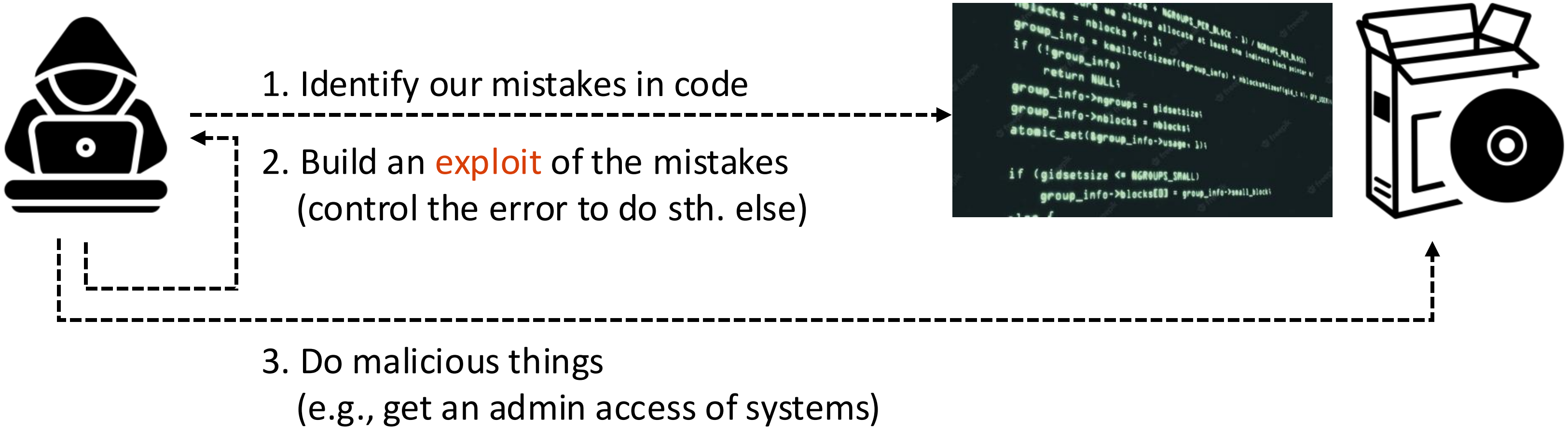
PROGRAM ANALYSIS

- Static analysis
  - Examine source code without running them
  - Search for the patterns that lead to security vulnerabilities – signatures
  - Use these signatures on the source code-level

- Dynamic analysis
  - Examine source code + running the code to observe program behaviors
  - Search for the (runtime) patterns that lead to security vulnerabilities – signatures
  - Use these signatures at runtime – help identify vulnerabilities that only manifest at runtime

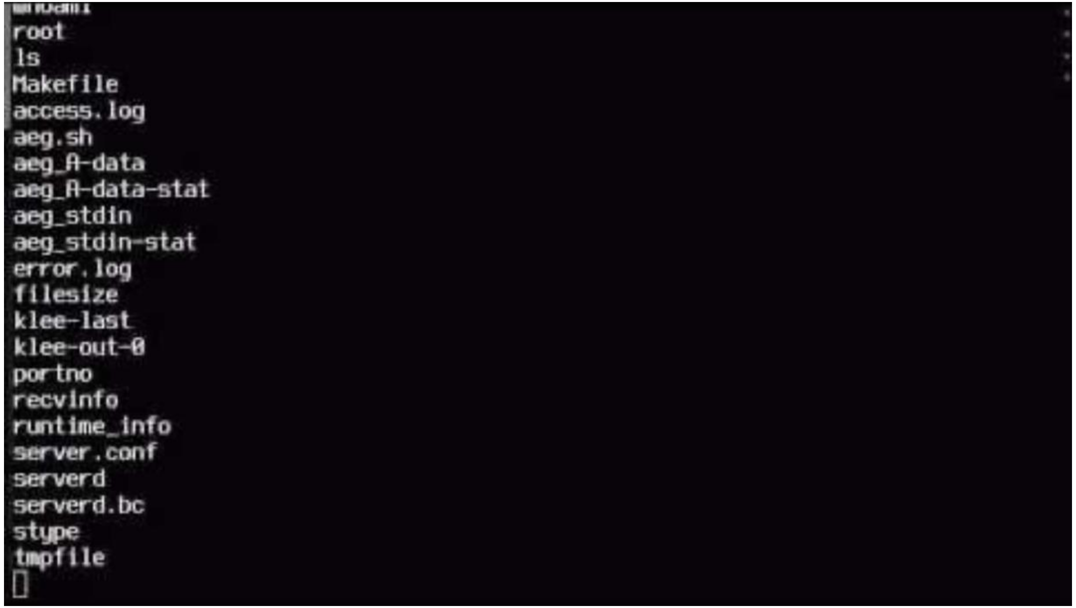
VULNERABILITY != EXPLOITATION

- Crash
- A hack



VULNERABILITY != EXPLOITATION

- Motivation
  - Given a program
    - Automatically find vulnerability
    - Generate exploits for them



PROGRAM ANALYSIS – AVG: MOSTLY STATIC

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)

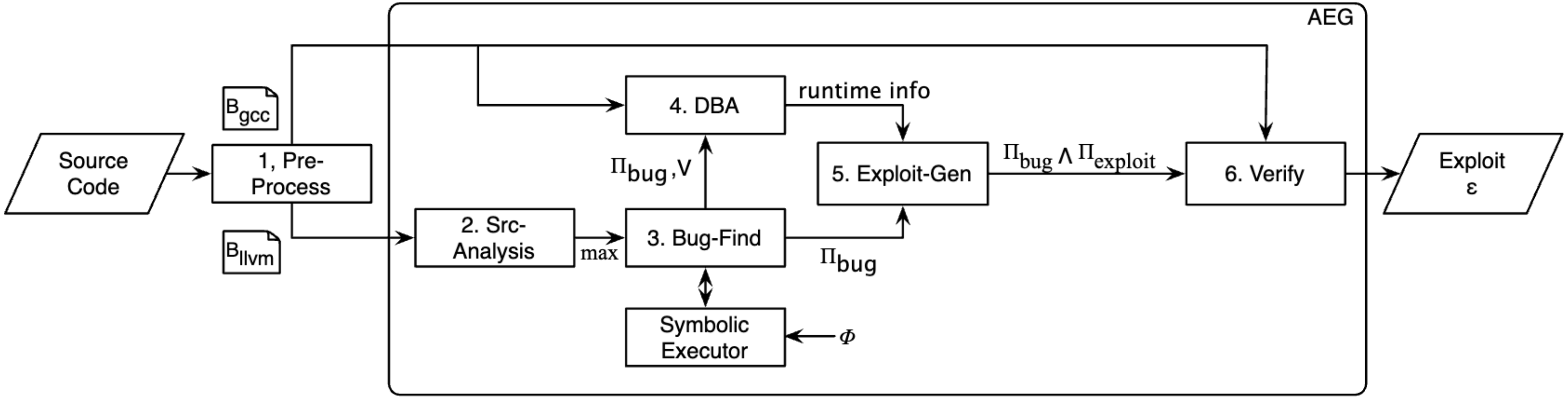


Figure 5: AEG design.

PROGRAM ANALYSIS – AEG: DYNAMIC ANALYSIS IS NEEDED

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
    - ex. iwconfig

```
int get_info(int skfd, char * ifname, …){
  ...
  if(iw_get_ext(skfd, ifname, SIOCGIWNAME, &wrq) < 0)
  {
    struct ifreq ifr;
    strcpy(ifr.ifr_name, ifname);
}

print_info(int skfd, char *ifname,…){
 ...
 get_info(skfd, ifname, …);
}

main(int argc, char *argv[]){
...
 print_info(skfd, argv[1], NULL, 0);
}
```

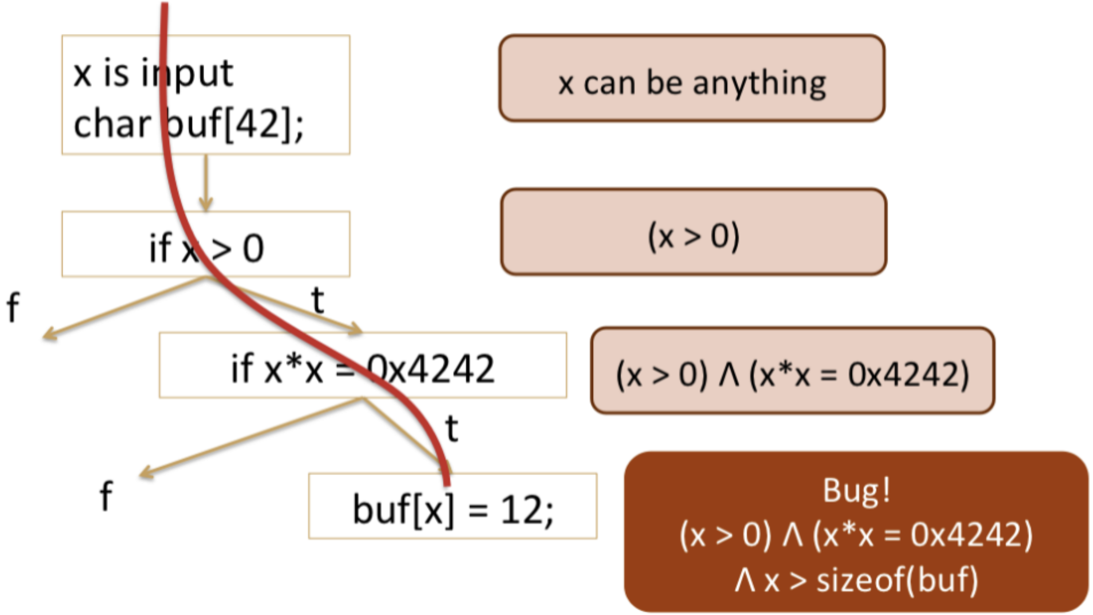
```
struct ifreq {
  char ifr_name[32]
  …
}
```

Can you spot the bug?

https://prezi.com/tf8tyeebha_8/aeg-automatic-exploit-generation/

PROGRAM ANALYSIS – AEG: DYNAMIC ANALYSIS IS NEEDED

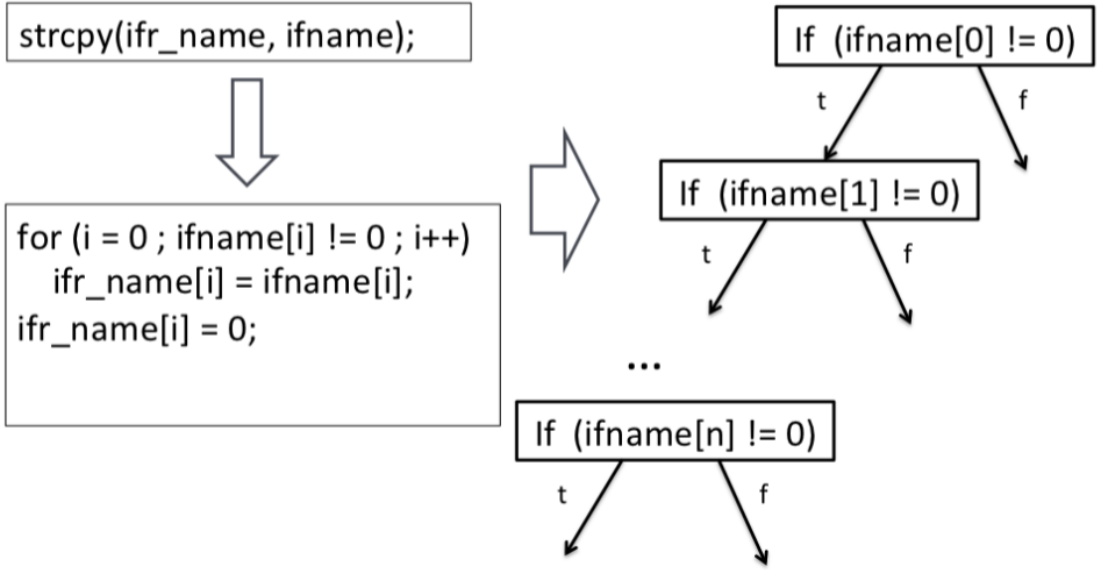
- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
    - ex. Iwconfig
    - Symbolic execution



https://prezi.com/tf8tyeebha_8/aeg-automatic-exploit-generation/

PROGRAM ANALYSIS – AEG: DYNAMIC ANALYSIS IS NEEDED

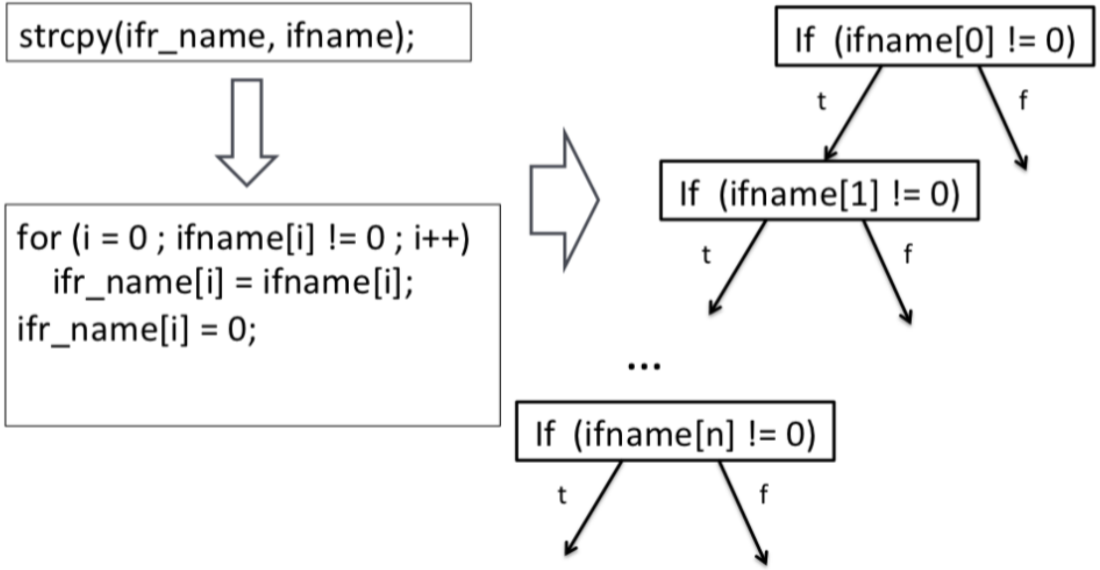
- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
    - ex. Iwconfig
    - Symbolic execution



https://prezi.com/tf8tyeebha_8/aeg-automatic-exploit-generation/

Program analysis – AEG: dynamic analysis is needed

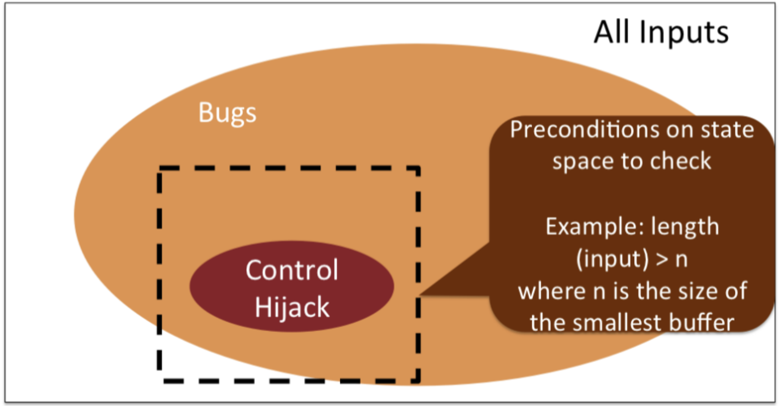
- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
    - ex. Iwconfig
    - Symbolic execution
    - Prior work: KLEE
      - Scalability issue
      - Need to prove the absence of bugs by running all the pa-ths



https://prezi.com/tf8tyeebha_8/aeg-automatic-exploit-generation/

PROGRAM ANALYSIS – AEG: DYNAMIC ANALYSIS IS NEEDED

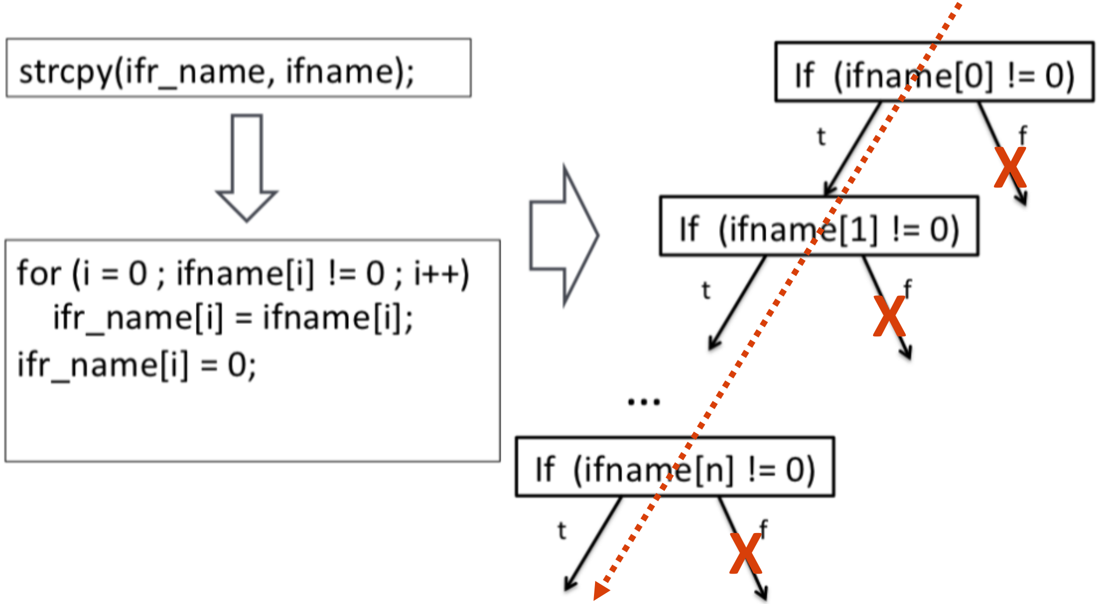
- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
    - Prior work: KLEE
      - Scalability issue
      - Need to prove the absence of bugs by running all the paths
    - This work: *precondition symbolic execution*



https://prezi.com/tf8tyeebha_8/aeg-automatic-exploit-generation/

PROGRAM ANALYSIS – AEG: DYNAMIC ANALYSIS IS NEEDED

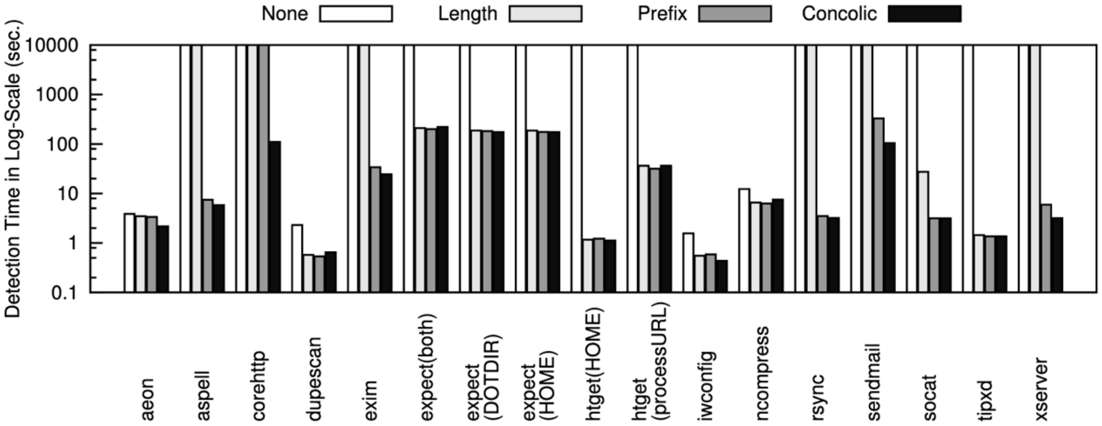
- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
    - ex. Iwconfig
    - Symbolic execution
    - Prior work: KLEE
    - This work: *precondition symbolic execution + path prior-ritization*



https://prezi.com/tf8tyeebha_8/aeg-automatic-exploit-generation/

PROGRAM ANALYSIS – AEG: DYNAMIC ANALYSIS IS NEEDED

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
    - ex. Iwconfig
    - Symbolic execution
    - Prior work: KLEE
    - This work: *precondition symbolic execution + path prior-ritization*



PROGRAM ANALYSIS – AEG: DYNAMIC ANALYSIS IS NEEDED

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
  - **DBA: Dynamic binary analysis**

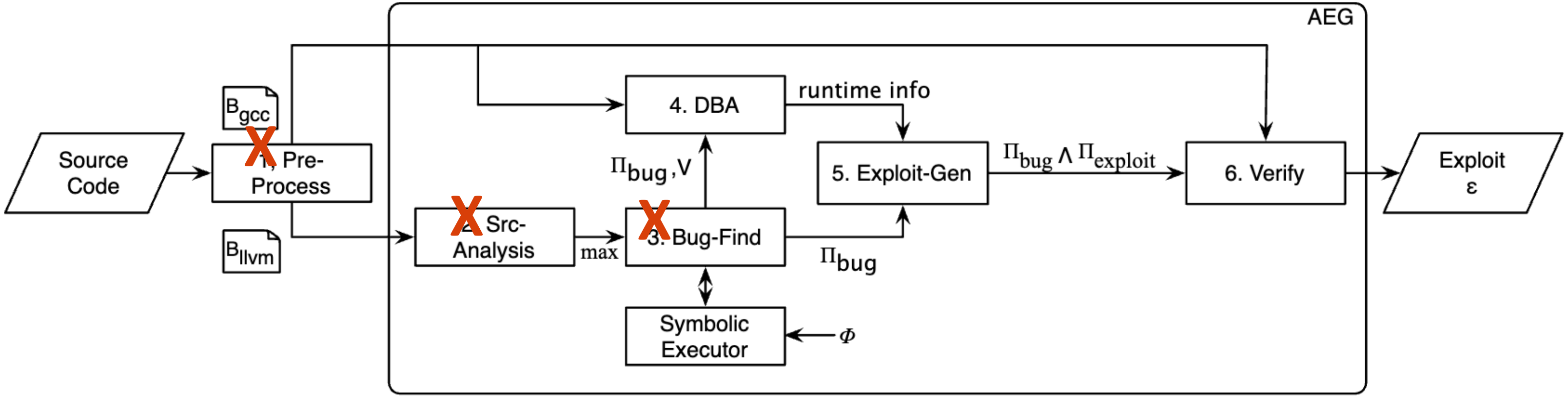


Figure 5: AEG design.

PROGRAM ANALYSIS – AEG: DYNAMIC ANALYSIS IS NEEDED

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
  - DBA
    - Information from the source code is insufficient for generating an exploit
    - Runtime information is needed
      - Think about your buffer overflow (HW 2)
      - The %rip address stored in memory when a function is called, is required

PROGRAM ANALYSIS – AEG: DYNAMIC ANALYSIS IS NEEDED

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
  - DBA
    - Information from the source code is insufficient for generating an exploit
    - Runtime information is needed
    - Process
      - Get a concrete input for the bugs generated by “Bug-Find”
      - Set a set of breakpoints om the vulnerable function (also found by “Bug-Find”)
      - Get runtime information, required to generate an exploit(s)

PROGRAM ANALYSIS – AEG: DYNAMIC ANALYSIS IS NEEDED

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
  - DBA
  - **Exploit-Gen**

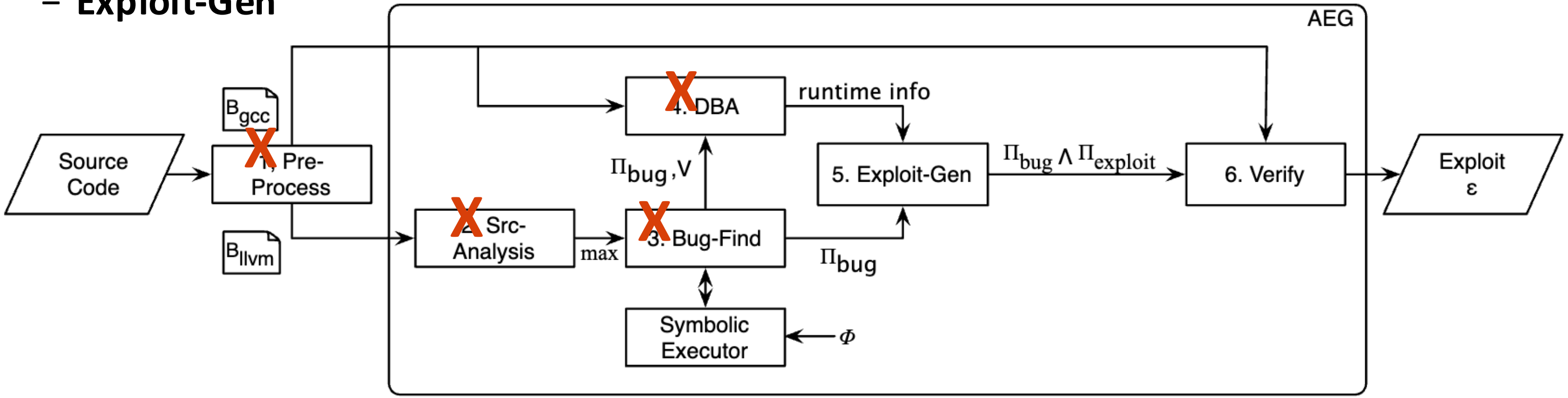


Figure 5: AEG design.

PROGRAM ANALYSIS – AEG: DYNAMIC ANALYSIS IS NEEDED

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
  - DBA
  - **Exploit-Gen**
    - Takes
      - Path constraints
      - Runtime information, e.g., vulnerable variable (buffer) address
      - Runtime stack memory constraints
    - Generates
      - Exploit formulas
        - ≫Stack overflow (return to libc and return to stack)
        - ≫Format string (return to libc and return to stack)

PROGRAM ANALYSIS – AEG: DYNAMIC ANALYSIS IS NEEDED

- Automatic exploit generation (AEG)
  - Preprocess the source code of a program
  - Program analysis (source code analysis)
  - Bug hunt
  - DBA
  - Exploit-Gen
  - **Verify**

	Program	Ver.	Exploit Type	Vulnerable Input src	Gen. Time (sec.)	Executable Lines of Code	Advisory ID.
None	aeon	0.2a	Local Stack	Env. Var.	3.8	3392	CVE-2005-1019
	iwconfig	V.26	Local Stack	Arguments	1.5	11314	CVE-2003-0947
	glftpd	1.24	Local Stack	Arguments	2.3	6893	OSVDB-ID#16373
	ncompress	4.2.4	Local Stack	Arguments	12.3	3198	CVE-2001-1413
Length	htget (processURL)	0.93	Local Stack	Arguments	57.2	3832	CVE-2004-0852
	htget (HOME)	0.93	Local Stack	Env. Var	1.2	3832	Zero-day
	expect (DOTDIR)	5.43	Local Stack	Env. Var	187.6	458404	Zero-day
	expect (HOME)	5.43	Local Stack	Env. Var	186.7	458404	OSVDB-ID#60979
	socat	1.4	Local Format	Arguments	3.2	35799	CVE-2004-1484
	tipxd	1.1.1	Local Format	Arguments	1.5	7244	OSVDB-ID#12346
Prefix	aspell	0.50.5	Local Stack	Local File	15.2	550	CVE-2004-0548
	exim	4.41	Local Stack	Arguments	33.8	241856	EDB-ID#796
	xserver	0.1a	Remote Stack	Sockets	31.9	1077	CVE-2007-3957
	rsync	2.5.7	Local Stack	Env. Var	19.7	67744	CVE-2004-2093
	xmail	1.21	Local Stack	Local File	1276.0	1766	CVE-2005-2943
Concolic	corehttp	0.5.3	Remote Stack	Sockets	83.6	4873	CVE-2007-4060
Average Generation Time & Executable Lines of Code					114.6	56784	

Thank You!

Sanghyun Hong

https://secure-ai.systems/courses/Sec-Grad/current

Oregon State University

SAIL

Secure AI Systems Lab