AEG: Automatic Exploit Generation

Benjamin Lim
(with some content shamelessly stolen from)
Wong Wai Tuck

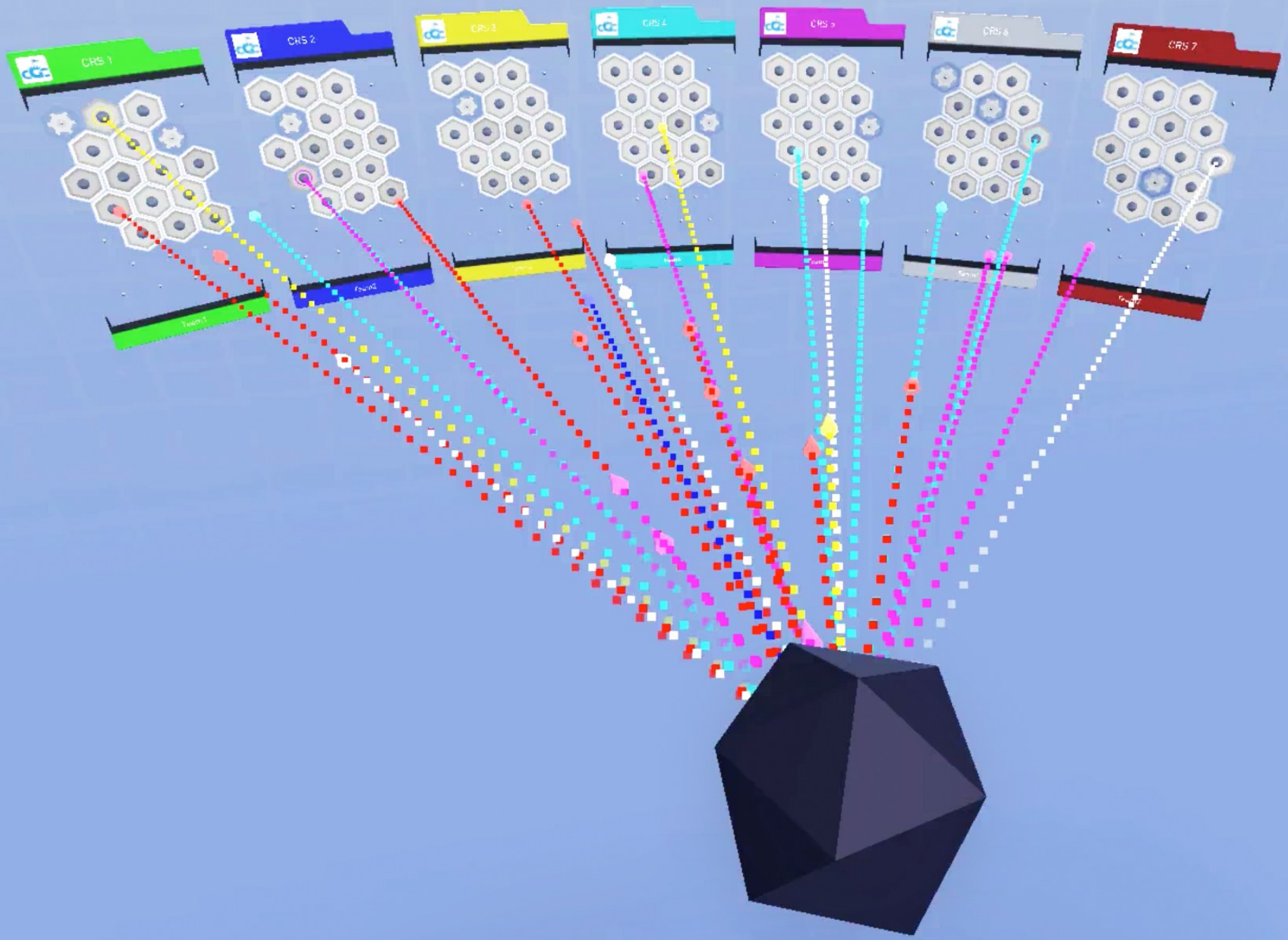
How do I pwn?

- Take a binary
- Find a vulnerability and inputs which trigger that vulnerability
- Create a payload which exploits the vulnerability
- ???
- ~~profit~~ responsible disclosure
- Vendor doesn’t patch it after several months
- profit

Why can’t I pwn?

- **Vulnerability discovery is a slow and tedious process**
- **Large size of binaries**
- **Vulnerability → Exploit can be nontrivial**
  - e.g. restrictions on input, insufficient space for shellcode, etc.
- **Patching of vulnerabilities varies in difficulty**


CGC
CYBER
GRAND_CHALLENGE

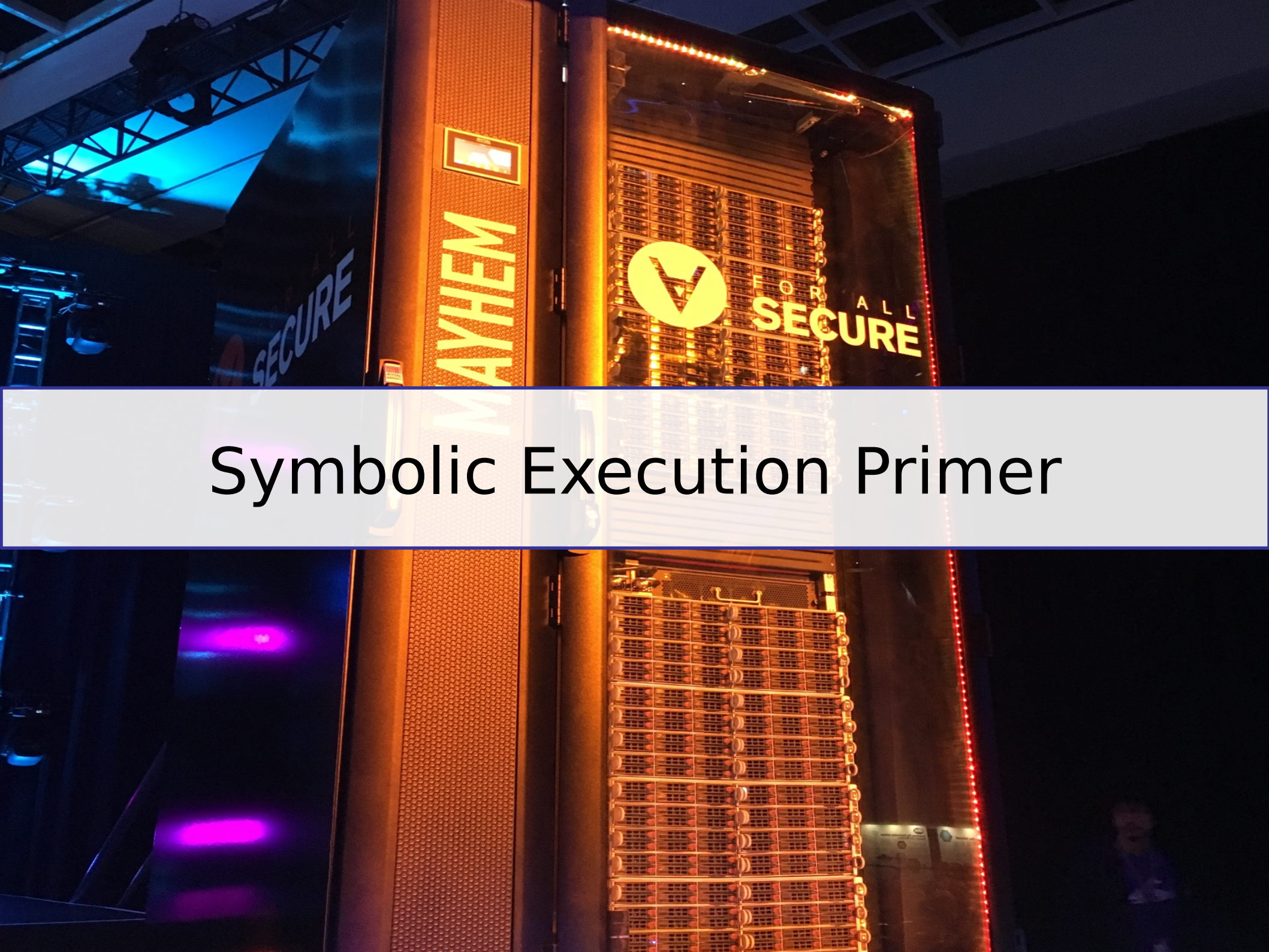

CRS 1
CRS 2
CRS 3
CRS 4
CRS 5
CRS 6
CRS 7

DARPA Cyber Grand Challenge (2014-)2016

- **Automatic exploitation and patching**
- **Custom pwnables written for DECREE OS**
- **DECREE caveats and rant**
- **Winners:**
  - 1st: Mayhem (CMU)
  - 2nd: Xandra (TECHx)
  - 3rd: Mechanical Phish (UCSB)
- **Only Mechanical Phish (angr) open-sourced :(**

Mayhem in CGC

- **Challenges modeled after real exploits**
  - Morris Worm (buffer overflow)
  - Stuxnet LNK (off by one) (CVE-2010-2568)
  - Crackaddr (buffer overflow) (CVE-2002-1337)
  - Heartbleed (leak of sensitive data) (CVE-2014-0160)
- **Patching**
  - Return pointer encryption
  - Protection of indirect calls/jmps
  - Extended malloc allocations
  - Manual ASLR
  - Cleaning of uninitialized space
- **For DEFCON challenge that was broadcasted at 14:10:25 UTC, hardened binary created at 14:11:08 UTC (43 seconds)**



Symbolic Execution Primer

Toy Program

```
void foo(int x, int y){
    int t = 0;
    if (x > y){
        t = x;
    } else{
        t = y;
    }

    if (t < x){
        assert false;
    }
}
```

Concrete Execution (testing)

```
void foo(int x, int y){
    int t = 0;
    if (x > y){
        t = x;
    } else{
        t = y;
    }

    if (t < x){
        assert false;
    }
}
```

- **Concrete Store:**
  - x = 4
  - y = 4

Concrete Execution (testing)

```
void foo(int x, int y){
    int t = 0;
    if (x > y){
        t = x;
    } else{
        t = y;
    }

    if (t < x){
        assert false;
    }
}
```

- **Concrete Store:**
  - x = 4
  - y = 4
  - t = 0

Concrete Execution (testing)

```
void foo(int x, int y){
    int t = 0;
    if (x > y){
        t = x;
    } else{
        t = y;
    }

    if (t < x){
        assert false;
    }
}
```

- **Concrete Store:**
  - x = 4
  - y = 4
  - t = 4

Concrete Execution (testing)

```
void foo(int x, int y){
    int t = 0;
    if (x > y){
        t = x;
    } else{
        t = y;
    }

    if (t < x){
        assert false;
    }
}
```

- **Concrete Store:**
  - x = 4
  - y = 4
  - t = 4
- **4 < 4 is false, quik maffs**

Concrete Execution (testing)

```
void foo(int x, int y){
    int t = 0;
    if (x > y){
        t = x;
    } else{
        t = y;
    }

    if (t < x){
        assert false;
    }
}
```

- **Concrete Store:**
  - x = 4
  - y = 4
  - t = 4
- **4 < 4 is false, assertion unreached**

Static Symbolic Execution

```
void foo(int x, int y){
    int t = 0;
    if (x > y){
        t = x;
    } else{
        t = y;
    }

    if (t < x){
        assert false;
    }
}
```

- **Symbolic Store:**
  - x = **X**
  - y = **Y**

Static Symbolic Execution

```
void foo(int x, int y){
    int t = 0;
    if (x > y){
        t = x;
    } else{
        t = y;
    }

    if (t < x){
        assert false;
    }
}
```

- **Symbolic Store:**
  - x = **X**
  - y = **Y**
  - t = 0

Static Symbolic Execution

```
void foo(int x, int y){
    int t = 0;
    if (x > y){
        t = x;
    } else{
        t = y;
    }

    if (t < x){
        assert false;
    }
}
```

- **Symbolic Store:**
  - x = **X**
  - y = **Y**
  - t = ite(**X<Y**,**X**,**Y**)

Static Symbolic Execution

```
void foo(int x, int y){
    int t = 0;
    if (x > y){
        t = x;
    } else{
        t = y;
    }

    if (t < x){
        assert false;
    }
}
```

- **Symbolic Store:**
  - x = **X**
  - y = **Y**
  - t = ite(**X<Y**,**X**,**Y**)
- Assert condition: ite(**X<Y**,**X**,**Y**)<**X**

Static Symbolic Execution

```
void foo(int x, int y){
    int t = 0;
    if (x > y){
        t = x;
    } else{
        t = y;
    }

    if (t < x){
        assert false;
    }
}
```

- **Symbolic Store:**
  - x = **X**
  - y = **Y**
  - t = ite(**X<Y,X,Y**)
- Assert condition: ite(**X<Y,X,Y**)<**X**
- **Throw into solver – assert not hit**

Dynamic Symbolic Execution

```
void foo(int x, int y){
    int t = 0;
    if (x > y){
        t = x;
    } else{
        t = y;
    }

    if (t < x){
        assert false;
    }
}
```

- **Symbolic Store:**
  - x = **X**
  - y = **Y**
  - T = 0
- Case split on conditional

Dynamic Symbolic Execution

```
void foo(int x, int y){
    int t = 0;
    if (x > y){
        t = x;
    } else{
        t = y;
    }

    if (t < x){
        assert false;
    }
}
```

- **Branch 1: X > Y**
- **Symbolic Store:**
  - x = **X**
  - y = **Y**
  - t = **X**
- Assert condition: **X<X**
- **Assert not hit**

Dynamic Symbolic Execution

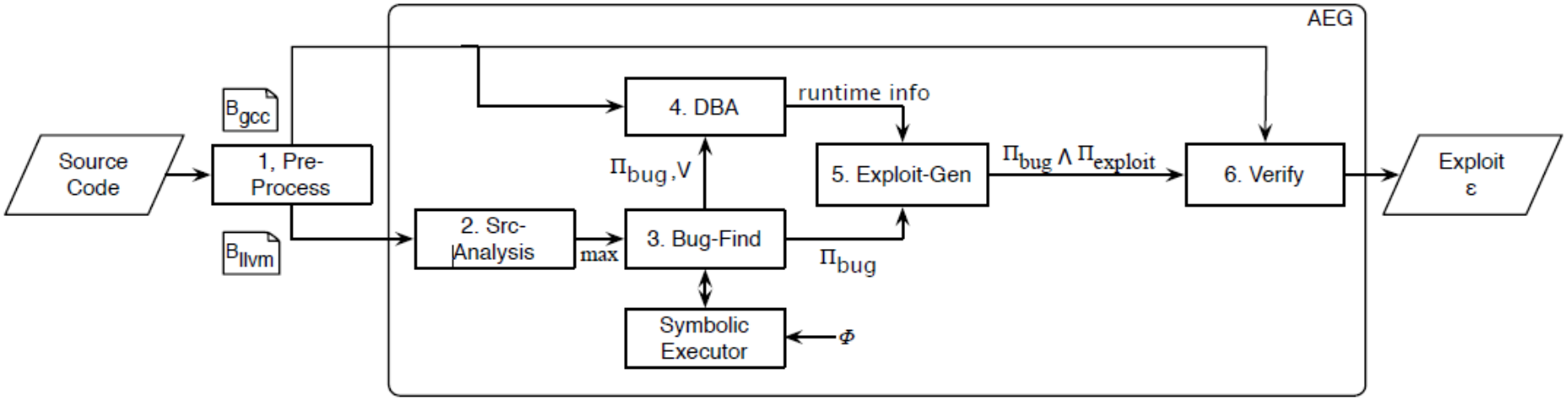
```
void foo(int x, int y){
    int t = 0;
    if (x > y){
        t = x;
    } else{
        t = y;
    }

    if (t < x){
        assert false;
    }
}
```

- **Branch 2: !(X > Y)**
- **Symbolic Store:**
  - x = **X**
  - y = **Y**
  - t = **Y**
- Assert condition: **Y<Y**
- **Assert not hit**

Actually Exploiting Stuff (kindof maybe)

AEG in Four Easy Steps

- **Symbolically execute program (warning! slow!)**
- **Detect violation of safety property**
- **Check if exploitable**
- **Generate exploit (using template shellcode)**



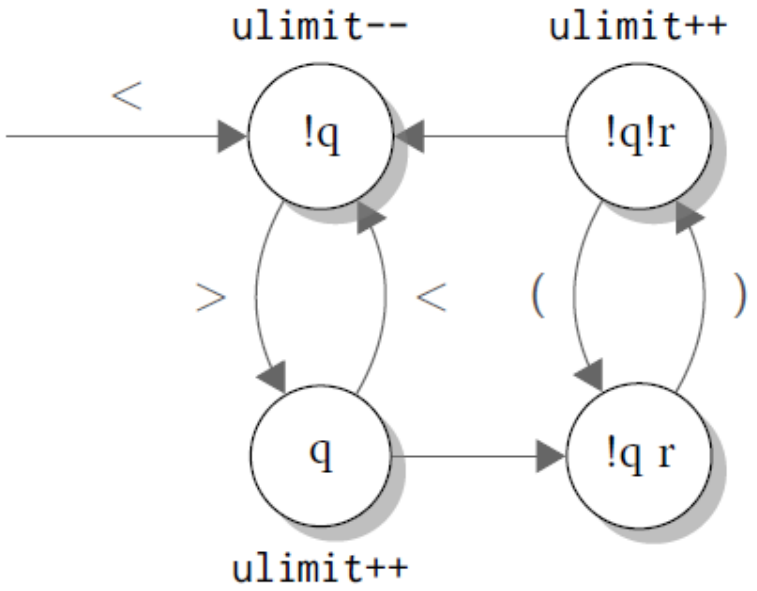
Case Study: Crackaddr Variant

- **CVE2002-1337**
  - Sendmail 5.79 to 8.12.7
  - Remote execution via buffer overflow in ‘crackaddr’ function of headers.c
- **CGC Challenge (Halvar Flake (2011))**
  - Extracted core of bug (50 LOC vs. 247)
  - ‘Tool should automatically show vulnerable version has a bug and the fixed version is safe’

Case Study: Crackaddr Variant

```
1  #define BUFFERSIZE 200
2  #define TRUE 1
3  #define FALSE 0
4  int copy_it (char *input, unsigned int length) {
5      char c, localbuf[BUFFERSIZE];
6      unsigned int upperlimit = BUFFERSIZE - 10;
7      unsigned int quotation = roundquote = FALSE;
8      unsigned int inputIndex = outputIndex = 0;
9      while (inputIndex < length) {
10         c = input[inputIndex++];
11         if ((c == '<') && (!quotation)) {
12             quotation = TRUE; upperlimit--;
13         }
14         if ((c == '>') && (quotation)) {
15             quotation = FALSE; upperlimit++;
16         }
17         if ((c == '(') && (!quotation) && !roundquote) {
18             roundquote = TRUE; upperlimit--; // decrementation was missing in bug
19         }
20         if ((c == ')') && (!quotation) && roundquote) {
21             roundquote = FALSE; upperlimit++;
22         }
23         // If there is sufficient space in the buffer, write the character.
24         if (outputIndex < upperlimit) {
25             localbuf[outputIndex] = c;
26             outputIndex++;
27         }
28     }
29     if (roundquote) {
30         localbuf[outputIndex] = ')'; outputIndex++; }
31     if (quotation) {
32         localbuf[outputIndex] = '>'; outputIndex++; }
33 }
```

its a state machine woaw



- 201 loop iterations to trigger bug
- 10 different paths through loop
- 5^{201} (approx 2^{664}) paths

Case Study: Unintended Solution

```
1  int copy( fileSystemType *fs, char *cmdline, unsigned int owner ) {
2    char sourcefile[FILENAME_SIZE];
3    char destfile[FILENAME_SIZE];
4    int x;
5
6    // skip over leading whitespace characters
7    while ( *cmdline != 0 && isspace(*cmdline) )
8      ++cmdline;
9
10   // if we hit the end of the line there were no filenames specified
11   if ( *cmdline == 0 ) {
12     return ERROR_BAD_PARMS;
13   }
14
15   x = 0;
16   while ( *cmdline != 0 && !isspace(*cmdline) ) {
17     if ( x < FILENAME_SIZE ) {
18       sourcefile[x] = *cmdline;
19     }
20
21     ++cmdline;
22     ++x;
23   }
24
25   sourcefile[x] = 0;
26   ...
27 }
```

Case Study: Unintended Solution

```
1  int main(void) {
2    char command[1024];
3    while (1) {
4      bzero(command, 1024);
5      getline(command, 1024);
6
7      i = 0;
8      while (command[i] != ' ' && i < strlen(command)) {
9        ++i;
10     }
11     command[i] = 0;
12
13     if ( strcmp(command, "list") == 0 ) {
14       ...
15     }
16     else if ( strcmp( command, "copy" ) == 0 ) {
17       retcode = copy( currentFS, command+i+1, currentUser );
18     }
19     ...
20   }
21 }
```

- **Solved by Mayhem (~1h 45m)**

oh no

- **Symbolic execution suffers from scaling issues**
- **Real world nuisances like libraries, device drivers, operating systems**
  - On top of standard binary analysis issues (e.g. CFG recovery)
- **A lot of effort has gone into making symbolic execution of programs more viable**

help! i’m too slow!

- **Handling path explosion**
  - Heuristic preconditions on state space
    - Known Length (automatic – max)
    - Known Prefix (manual, e.g. HTTP GET)
    - Concolic Execution (manual, crashing input)
  - Heuristic path prioritization
    - Buggy-path-first
    - Loop Exhaustion

help! i’m too slow!

- **Handling state space explosion**
  - ‘Driller’ architecture (Mechanical Phish)
    - Dynamic Symbolic Execution with fuzzing
    - Each shores up weaknesses of the other
  - Veritesting (CMU Cylab)
    - Alternate between dynamic and static symbolic execution
    - Balances between the solver and the symbolic execution engine

help! the real world exists!

- **Handling the real world**
  - Actually symbolically execute into kernel/library
    - (probably going to fail)
  - Function/syscall hooking
    - Unconstrained symbolic values
    - Model effects of function call on symbolic state
    - Tedious and possibly error prone

help! the real world exists!

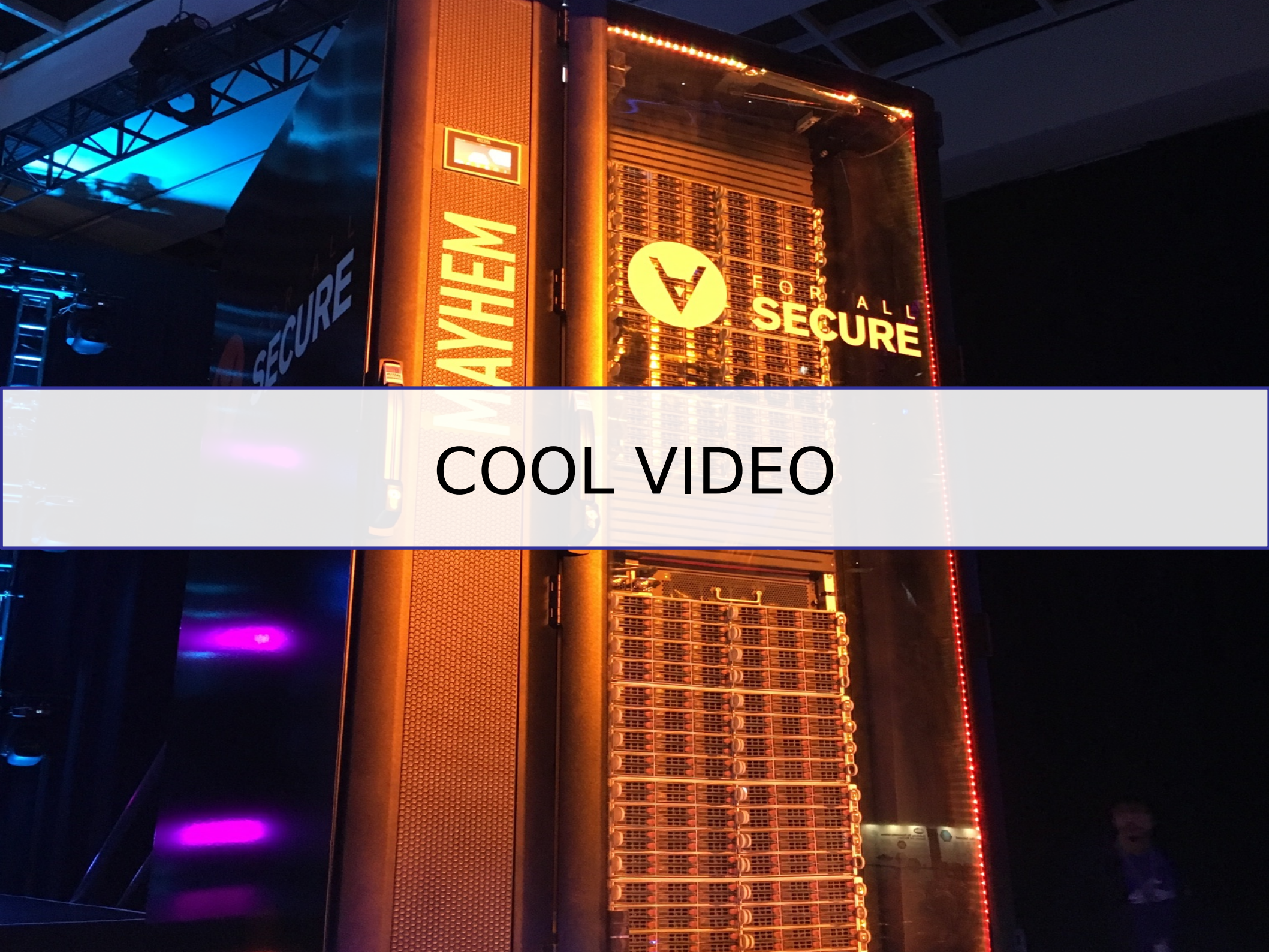
- **Handling the real world**
  - Indirect jumps/calls
    - Resolve all jump targets
    - Randomly concretize
  - S2E framework (‘in-vivo’ execution)
    - Switch between concrete and symbolic execution
    - Concretize e.g. syscall inputs, make symbolic after return

Some Remarks

- **AEG is a relatively new and developing field**
- **Techniques have been around for decades**
- **Practical implementations of AEG are still very much in development**
- **Real world is hard**
- **Formal methods is (are?) cool**

Useful Readings

- **Symbolic Execution Survey**
  - https://github.com/season-lab/survey-symbolic-execution
- **Decision Procedures, SMT solving**
  - The Calculus of Computation (Bradley, Manna)
  - Logic in Computer Science (Huth)
- **Theorem Proving/Provers**
  - CPDT (Chlipala), DeepSpec project
  - CompCert, seL4
  - Coq, Isabelle/HOL, Twelf, Idris, etc. etc.



COOL VIDEO

Cool video

- D:\Documents\AEG Exploits Demo.mp4

thanken you

qeustions?