# Teaching with `angr`: A Symbolic Execution Curriculum and CTF[*]

*Jacob M. Springer    Wu-chang Feng*
*Portland State University*
*Department of Computer Science*

## Abstract

Symbolic execution is an essential tool in modern program analysis and vulnerability discovery. The technique is used to both find and fix vulnerabilities as well as to identify and exploit them. In order to ensure that symbolic execution tools are used more for the former, rather than the latter, we describe a curriculum and a set of scaffolded, polymorphic, "capture-the-flag" (CTF) exercises that have been developed to help students learn and utilize the technique to help ensure the software they produce is secure.

## 1 Introduction

Software flaws and vulnerabilities are inevitable, with one occurring in approximately every 100 lines of code written. With the sheer amount of code being produced, it is becoming exceedingly difficult to secure the software we rely upon. The fact that flawed source code is often shared among many projects exacerbates the impact of any underlying vulnerabilities. Unfortunately, developers properly trained in security, though increasingly more important to software development, are scarce. To address this problem, there has been increasing interest in automating the process of vulnerability discovery and patching. Such was the goal of DARPA's recent Cyber Grand Challenge (CGC), a contest where computers were tasked with automatically analyzing, exploiting, and ameliorating vulnerabilities in arbitrary binaries. One of the key techniques used by all of the teams was symbolic execution, significantly reducing the time required to effectively analyze code paths within a target binary.

Since it is possible now for adversaries to employ symbolic execution to find vulnerabilities, it is critical that software developers master the technique as well. Unfortunately, the concepts and tools surrounding symbolic execution are rarely taught in computer science curricula, leaving many students at a disadvantage when attempting to write vulnerability-resistant software. To address this problem, this paper describes and evaluates a curriculum for teaching the concepts of symbolic execution along with a scaffolded, polymorphic set of Capture-the-Flag (CTF) challenges. The curriculum and CTF are freely available in order to enable computer science programs across the country to teach these techniques to students within their security and software engineering courses.

## 2 Background

Symbolic execution has fundamentally changed how programs are now being tested. Analogous to solving algebraic expressions, symbolic execution explores possible code execution paths until it identifies a potential vulnerability, at which point it attempts to solve for the particular input that will exploit the vulnerability. More specifically, a symbolic execution engine replaces input with "symbolic input"—analogous to an algebraic variable—and walks through code paths, "constraining" the symbolic input at each branch such that an input to the program that satisfies all constraints will cause the program to reach that particular path. The engine can then explore many possible execution paths until it identifies a specific path or program state of interest, at which point it can determine the input which would trigger it.

One of the key applications for symbolic execution

[*]This material is supported by the National Science Foundation under Grant No. 1623400. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

```
1      if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
2          goto fail;
3      if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
4          goto fail;
5      if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
6          goto fail;
7          goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
8      if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
9          goto fail;
10
11     err = sslRawVerify(...);
```

Figure 1: Apple's `goto fail` bug

is to catch programming errors. For example, Figure 1 shows the bug in Apple's SSL implementation in which a duplicate `goto fail;` statement in line 7 was injected into code, allowing one to bypass certificate validation with a carefully crafted certificate. In the figure, the duplicate `goto fail;` is always executed, and, as a result, the code between it and the `fail:` label is unreachable. Symbolic execution can be used to catch such an error by automatically discovering that no input will allow the program to reach the `SSLRawVerify()` call in line 11.

Two key issues with symbolic execution prevent symbolic execution engines from running efficiently. First, executing programs symbolically can lead to state-space explosion as the engine can encounter too many possible execution paths to check in a reasonable amount of time. Second, the engine may place arbitrary constraints on variables and thus satisfying all of them is an instance of satisfiable modulo theories (SMT), which is NP-complete. As a result, symbolic execution engines can borrow from fuzzers in selectively choosing to concretize certain input that is deemed uninteresting in order to save execution overhead. This technique, known as concolic execution, can be extremely powerful and was used in the fast and automatic discovery of the `crackaddr` vulnerability [1]. One of the most prominent early examples of concolic testing was SAGE from Microsoft, which is used internally to help secure the Windows operating system since Windows 7. Open-source versions such as KLEE [2] and S2E [3] are now widely used in industry and academia to discover and patch security vulnerabilities [4, 5].

## 3 Curriculum and CTF

In order to facilitate the teaching and training of students in applying symbolic execution on program binaries, we have developed a curriculum and set of scaffolded CTF challenges. The curriculum consists of 4 modules offered in 4 classes, each an hour and 50 minutes long. Students are introduced to specific concepts in a lecture format and immediately follow it by attempting to solve levels linked to each concept presented. The symbolic execution CTF exercises leverage MetaCTF, a prior, CTF designed for teaching malware reverse-engineering [6]. In MetaCTF, reverse engineering is used to determine a password that, when entered, causes the level binary to output the string `"Good Job."`. Such a construction is helpful since it allows students to focus on a specific goal that does not change for each level.

Adapting this level design, an analogous set of CTF levels that target the teaching of symbolic execution was designed. The CTF is based upon `angr` [7], an open-source symbolic execution engine from University of California, Santa Barbara. The CTF currently consists of 18 polymorphically generated levels that require students to apply symbolic execution in a variety of ways in order to solve. Specifically the CTF levels task students with writing Python programs using `angr` that load the binary and symbolically execute it in order to identify the password required to unlock the level. For each level, students are given a template Python script with key parts missing that they must fill in to allow `angr` to automatically execute and discover the input required to solve each of the polymorphic binaries. The templates contain a detailed description of what students need to implement so that they can focus on the specific concept that the level is attempting to cover. Each template is associated with a specific polymorphic binary. The binaries

```c
int check_code(int input){
   if (input >= USERDEF+88) return 0;
   if (input > USERDEF+100) return 0;
   if (input == USERDEF+68) return 0;
   if (input < USERDEF) return 0;
   if (input <= USERDEF+78) return 0;
   if (input & 0x1) return 0;
   if (input & 0x2) return 0;
   if (input & 0x4) return 0;
   return 1;
}

int main (int argc, char** argv) {
   int input;
   scanf("%d", &input);
   if (check_code(input))
     printf("Good Job.\n");
   else {
     printf("Try again.\n");
   }
}
```

Figure 2: C source code for MetaCTF level used to introduce symbolic execution

themselves are constructed in a way that makes manual reverse-engineering infeasible while at the same time, in a way that allows symbolic exeuction to solve in a modest amount of time. This is done by taking user input and performing a simple hash function on it before passing it into the level. By completing the CTF, students will have gained sufficient experience and skills with symbolic execution to then apply them to more complex cases. In the following section, we describe each of the 4 modules.

## 3.1 Basic symbolic execution

The initial module and CTF levels cover basic symbolic execution techniques including the algorithms being used to perform the execution and the abstractions `angr` uses to access program execution. Within the introductory lecture, a simple example is used to show how symbolic execution works. The level, whose C code is shown in Figure 2, is taken directly from a MetaCTF level that students have solved earlier in the course whose goal is to teach them how to decode conditional branches. In the original level, students are only given the compiled binary. Within the binary, each of the comparisons in the C program is implemented with a variety of conditional branches. By manually keeping track of the contraints that allow the `check_code` function to eventually return 1 and cause the program to print `"Good Job."`, students then calculate the solution. Using this level, we simulate visually how symbolic execution would keep track of all paths through the program, executing each and splitting off states at each conditional branch while updating the constraints in each branch based on the path through the program. When symbolic execution reaches the desired outcome state, the constraints it has accumulated are exactly the same as the constraints students had kept when manually solving the level via reverse-engineering. However, instead of manually solving the constraints, the symbolic execution engine sends them to a constraint solver to *automatically* determine the input that satisfies them (if any).

```c
int main(int argc, char* argv[]) {
   char buf[9];

   printf("Enter the password: ");
   scanf("%8s", buf);

  for (int i=0; i<LEN_USERDEF; ++i)
     buf[i] = complex_function(buf[i], i)
    ;

  if (strcmp(buf, USERDEF))
     printf("Try again.\n");
  else
     printf("Good Job.\n");
}
```

Figure 3: C code for first CTF level

After the initial introduction, we task students with solving their own polymorphically-generated levels using symbolic exeuction. The first level has students use the `find` method in `angr` as well as its support for automatically making standard input symbolic, in order to have the engine find an input that reaches a specific line of code in a program. Figure 3 shows the source code for the level. Students are then given the compiled binary and asked to find the input that causes `"Good Job."` to be printed. While they could manually reverse-engineer the binary, as shown in the figure, the input they send in is sent through a "complex function" to make reverse-engineering the level more difficult than solving the level with an `angr` script. Instead, students bring up the level in a debugger or disassembler such as `radare2`, in order to find the line of the binary that they will point `angr` to in order to find the input that reaches it. For example, Figure 4 shows the virtual address for the level that contains the code for printing the `"Good Job."` string (`0x0804867a`). Students simply use this address in the `find` method and then run the engine in order to reveal

```
0x804867a ;[gi]
  sub esp, 0xc
     ; 0x8048760
     ; "Good Job."
  push str.Good_Job.
  call sym.imp.puts;[gk]
  add esp, 0x10
```

Figure 4: `radare2` disassembly for first CTF level

the solution. Figure 5 shows the `angr` script which can solve the level automatically once students fill in the virtual address identified in `radare2`, thus allowing `angr` to do the work for them.

In addition to using `angr` to find an input that results in a specific program address to be reached, levels in the introductory module also have students apply the `avoid` condition to automatically terminate execution paths that do not lead to useful states. This is necessary in some cases because symbolic execution can otherwise become prohibitively expensive due to state explosion. Finally, while finding an input that causes the program to reach a specific instruction is one way to apply symbolic execution, one can create arbitrary conditions for `angr` to either terminate execution or to return a solution. One particularly helpful condition that can be used is to find program input that leads to specific program output. This is appropriate in our levels since each level either outputs `"Good Job."` or `"Try again."`. In this case, students learn how to implement find and avoid states and apply them to standard output to get the program to print `"Good Job."` and avoid `"Try again"`.

## 3.2 Symbol injection

For some binaries, only a subset of a program can be symbolically executed. For example, one might wish to symbolically execute a device driver within the kernel rather than the entire operating system. To support this, a common method in symbolic execution is to concretely execute until a certain point, then insert symbolic values for any number of desired locations. Note that while previous levels take advantage of the fact that standard input is automatically made symbolic, typically symbols must be injected manually.

In the second module of our CTF, levels that require students to programmatically inject symbols into program simulation are given. To facilitate this, students are introduced to the concept of program states and how `angr` can be used to concretely execute the binary until a certain point in the program is reached. Students are then introduced to the data structures used to instantiate symbols and to inject them into the program's execution state. Specifically, the concept of bitvectors as implemented via Claripy is introduced and example scripts are shown that use the facility. With this, students are given levels in which they must inject symbols into execution state to solve. In one level, they replace specific registers with symbols. In other levels, they replace variables stored either in statically allocated global memory, in dynamically allocated memory in the heap, or in specific locations in the stack. Finally, they are given a level in which the contents of a file are treated as symbolic.

Figure 6 shows the disassembly of the level in which students are asked to start symbolic execution in the middle of a program after making a set of registers symbolic. In the level, a call to the function `get_user_input` is made in whicn values are read into three different registers. Students inject symbols into each register and start symbolic execution right after the return from the function (`0x80488d1`). Figure 7 shows part of the `angr` script that implements the solution. As the script indicates, after specifying the start address for symbolic execution, Claripy bit-vectors are declared and injected into the execution state. Conditions are then defined for success and failure before the execution engine is invoked to solve the level.

## 3.3 Constraints and function hooks

The third module introduces constraints and function hooks for reducing the state-space being searched in addition to reducing the complexity of well-known functions. Adding contraints on symbols is similar to avoiding certain execution states as described previously. When one can partially constrain certain symbols during exeuction, effectively concretizing certain symbolic input, it allows the symbolic execution engine to focus in on inputs that are more interesting. To practice applying contraints, the third module includes a level in which a constraint can be applied during a specific part of execution that, when met, will allow execution to return a solution without further execution.

Another way of addressing this problem is to replace complex, but well-known functions, with a simplified equivalent. For example, rather than symbolically execute the complex standard C library routines which is expensive and potentially unnecessary, `angr` automatically hooks many of them and instead, replaces them

```
import angr
import sys

def main(argv):
  path_to_binary = argv[1]
  project = angr.Project(path_to_binary)
  initial_state = project.factory.entry_state()
  simulation = project.factory.simgr(initial_state)

  print_good_address = 0x804867a
  simulation.explore(find=print_good_address)

  if simulation.found:
    solution_state = simulation.found[0]
    print solution_state.posix.dumps(sys.stdin.fileno())
  else:
    raise Exception('Could not find the solution')
```

Figure 5: angr solution script for first CTF level

```
0x080488cc   call sym.get_user_input
0x080488d1   mov dword [local_14h], eax
0x080488d4   mov dword [local_10h], ebx
0x080488d7   mov dword [local_ch], edx
0x080488da   sub esp, 0xc
0x080488dd   push dword [local_14h]
```

Figure 6: radare2 disassembly for symbolic register CTF level

with a simplified summary routine that executes concretely. In order to teach students how to leverage hooks, levels that require them to implement hooks within the symbolic execution engine in order to replace parts of the program being analyzed with simpler summaries are given. Figure 8 shows a snippet from a level in which students replace complex calls in the Standard C library with simpler equivalents supplied by angr that are written in Python and executed concretely. In doing so, the level can be solved in significantly less time. Note that while this level could be solved by brute-force, students often insisted on solving the level the intended way in order to learn what the level was attempting to teach them.

## 3.4 Binary exploitation

The last module caps the curriculum and CTF by having students apply angr to automatically find input that subverts a vulnerable binary and relies upon concepts and techniques in the previous three modules. The first set of levels have students identify vulnerable reads and writes in a program that can be exploited. Memory leaking such as with the Heartbleed vulnerability and memory corruption such as with procedure-link table (PLT) hijacking can occur when adversarial input controls a pointer that is used to either read from or write to memory. To identify this vulnerability, the symbolic execution engine can specifically look for unconstrained or symbolic memory reads and writes, that is, when a memory addressing mode uses a register that is symbolic to either read from or write to memory. In a first step towards leveraging symbolic execution to automatically exploit vulnerabilities, levels that teach students how to set up the engine to look for unconstrained memory access and how to then constrain the access to target specific locations for exploitation are used to solve levels of the CTF.

The final level serves as the capstone to the CTF. In the level, students find and exploit an unconstrained execution state. Specifically, students use symbolic execution to automatically identify and exploit a vulnerability that allows the input to control a return address stored on the stack. By pointing that return address to a function of their choosing, they can then manipulate the binary's code execution. Consider the source code for the final level of the CTF shown in Figure 9. As the figure shows, in normal operation, the program prints "Try again." and exits. However, a buffer overflow exists in the function read_input that can be leveraged to perform a return-oriented exploit that allows the "print_good" call to execute. One can use symbolic execution to discover the buffer overflow vulnerability exists because when the symbolic input is

```
start_address = 0x80488d1
initial_state = project.factory.blank_state(addr=start_address)

password0 = claripy.BVS('password0', 32)
password1 = claripy.BVS('password1', 32)
password2 = claripy.BVS('password2', 32)

initial_state.regs.eax = password0
initial_state.regs.ebx = password1
initial_state.regs.edx = password2

simulation = project.factory.simgr(initial_state)

def is_successful(state):
  stdout_output = state.posix.dumps(sys.stdout.fileno())
  return 'Good Job.' in stdout_output

def should_abort(state):
  stdout_output = state.posix.dumps(sys.stdout.fileno())
  return 'Try again.' in stdout_output

simulation.explore(find=is_successful, avoid=should_abort)
```

Figure 7: `angr` solution script template for symbolic register CTF level

```
initial_state = project.factory.entry_state()

project.hook(0x804ed40, angr.SIM_PROCEDURES['libc']['printf']())
project.hook(0x804ed80, angr.SIM_PROCEDURES['libc']['scanf']())
project.hook(0x804f350, angr.SIM_PROCEDURES['libc']['puts']())
project.hook(0x8048d10, angr.SIM_PROCEDURES['glibc']['__libc_start_main']())

simulation = project.factory.simgr(initial_state)

simulation.explore(find=is_successful, avoid=should_abort)
```

Figure 8: `angr` solution script for SimProcedure CTF level

made much larger than the size of the local buffers in the `read_input` routine, the address that contains the return address of the function becomes symbolic. When the return associated with the function is then executed, it will then generate an unconstrained state. That is, an input has been found that has led the engine to a state in which the instruction pointer itself is symbolic and can take on *any* value. Finding this unconstrained state, however, doesn't immediately lead to a level solution. In the case of this particular level, we wish to find an input that will lead to a return to the function "`print_good`". To do so, we need to find its address in assembly, then constrain the unconstrained instruction pointer to it, before invoking the solver to generate the actual input that will lead to the execution of "`print_good`".

To do so, students first must use a debugger or disassembler to find the address of "`print_good`" as shown in Figure 10. Then, they are tasked with adapting an `angr` script that attempts to find an input that will cause "`print_good`" to be executed. Note that there will be a very large number of potential solutions since all of the bytes used to perform the overflow, aside from the ones written to the return address on the stack can be arbitrary. Figure 11 shows part of the `angr` script students write to find a solution to the level. As the figure shows, the symbolic simulation is continuously stepped until an unconstrained execution state is discovered. When one is found, the instruction pointer is constrained to be the address of the "`print_good`" function before given to the solver to produce the input which exploits the vulnerability to solve the level.

```c
void print_good() {
  printf("Good Job.\n");
  exit(0);
}

void read_input() {
  char padding0[RND1];
  char buffer[8];
  char padding1[RND2];
  scanf("%s", buffer);
}

int main(int argc, char* argv[]) {
  uint32_t key = 0;

  printf("Enter the password: ");
  read_input();

  printf("Try again.\n");
  return 0;
}
```

Figure 9: C code for unconstrained jump CTF level

```
sym.print_good ();
        0x4d4c4749      push ebp
        0x4d4c474a      mov ebp, esp
        0x4d4c474c      sub esp, 8
        0x4d4c474f      sub esp, 0xc
        0x4d4c4752      push str.Good_Job
    .
        0x4d4c4757      call sym.imp.puts
```

Figure 10: `radare2` disassembly for unconstrained jump CTF level

## 4 Evaluation

The first offering of our symbolic execution CTF based on `angr` occurred in our Winter 2018 offering of Portland State University's CS 492/592 Malware course. The first 8 weeks of the course features a curriculum focused on malware analysis and reverse engineering using a variety of techniques covering both static and dynamic analysis. Students familiarize themselves with both Windows and Linux tools analyzing binaries such as IDA Pro and `radare2` with a curriculum that aligns with the first 18 chapters of the course's textbook [8]. Throughout this time, homework assignments are given via the scaffolded, metamorphic CTF described previously [6]. Note that, as a direct result of our plan to introduce symbolic execution with `angr` in the final two weeks of the class, all of the CTF levels of the malware reverse-engineering CTF had to be modified to resist symbolic execution. Without these mechanisms, a single symbolic execution script was able to to solve almost half of the binaries in this CTF.

| Completion percentage | No. of students |
|---|---|
| 95-100% | 25 |
| 85-95% | 4 |
| 75-85% | 6 |
| Below 75% | 7 |

Table 1: Level completion results

In the final two weeks, all 4 modules of the symbolic execution curriculum and CTF were covered. For each, 2 hour class, the first 30-45 minutes were spent introducing the concepts via lecture while the remaining time was spent by students attempting to solve their individual levels. At the beginning of each class, a set of hints were given on earlier levels in order to keep students from falling behind.

Table 1 lists the number of students who completed a certain percentage of assigned levels. The majority of students solved all of the levels indicating the curriculum and the scaffolding of the CTF work well. To provide a subjective measure of assessment, upon completion of the material, an anonymous survey was given. Of the 42 students in the class, 33 responded. Table 2 lists the questions that were asked in the survey, while Table 3 shows the results. As the table shows, students felt that the lecture material and CTF exercises were helpful in learning about symbolic execution and developing skills to apply it. Compared to other methods used in the homework for the course, the CTF exercises were also favored.

## 5 Conclusion

Symbolic execution is an important tool for ensuring that the software we develop is free of bugs and vulnerabilites. This paper describes a curriculum and CTF for not only teaching symbolic execution to students, but also developing their skills in applying it to program binaries. Results from an initial offering are promising and the curriculum [9], along with a hosted site containing the CTF [10], are both publicly available.

## References

[1] DARPAtv, "DARPA's Cyber Grand Challenge: Final Event Program," 2017, `https://www.youtube.com/watch?v=n0kn4mDXY6I`.

```
while (has_active() or has_unconstrained_to_check()) and (not has_found_solution()):
  for unconstrained_state in simulation.unconstrained:
    simulation.move('unconstrained', 'found')
  simulation.step()

if simulation.found:
  solution_state = simulation.found[0]
  solution_state.add_constraints(solution_state.regs.eip == 0x4d4c4749)

  solution = solution_state.posix.dumps(sys.stdin.fileno())
```

Figure 11: `angr` solution script for unconstrained jump CTF level

| Question |
|---|
| Q1: Rate the lecture material for understanding the concepts behind symbolic execution. |
| Q2: Rate the CTF exercises for understanding the concepts behind symbolic execution. |
| Q3: Rate the CTF exercises for developing skills in using symbolic execution techniques. |
| Q4: Helpfulness of the CTF format compared to other homework formats used in our curriculum |

Table 2: Survey questions for the `angr` CTF and curriculum in CS 492/592: Malware (Winter 2018)

| Question | 1 | 2 | 3 | 4 | 5 | Mean rating |
|---|---|---|---|---|---|---|
| Q1 | 1 | 1 | 2 | 17 | 12 | 4.15 |
| Q2 | 2 | 1 | 3 | 18 | 9 | 3.94 |
| Q3 | 1 | 3 | 3 | 16 | 10 | 3.94 |
| Q4 | 1 | 3 | 12 | 12 | 5 | 3.52 |

Table 3: Quality and Usefulness of Symbolic Execution module (1=Very unhelpful, 2=Somewhat unhelpful, 3=neither helpful nor unhelpful, 4=somewhat helpful, 5=Very helpful)

[2] The KLEE Team, "KLEE LLVM Execution Engine," `https://klee.github.io/`.

[3] "S2E: A Platform for In-Vivo Analysis of Software Systems," `https://s2e.systems/`.

[4] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "POTUS: Probing Off-The-Shelf USB Drivers with Symbolic Fault Injection," in *USENIX Workshop on Offensive Technologies*, August 2017.

[5] S. Kim, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim, "CAB-Fuzz: Practical Concolic Testing Techniques for COTS Operating Systems," in *USENIX Annual Technical Conference*, July 2017.

[6] W. Feng, "A Scaffolded, Metamorphic CTF for Reverse Engineering," in *USENIX 3GSE*, August 2015.

[7] "angr, a binary analysis framework," `http://angr.io/`.

[8] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, No Starch Press, 2012.

[9] W. Feng, "CS 492/592: Malware," `http://thefengs.com/wuchang/courses/cs492`.

[10] W. Feng, "CS 492/592: Malware CTF site," `https://malware.oregonctf.org`.