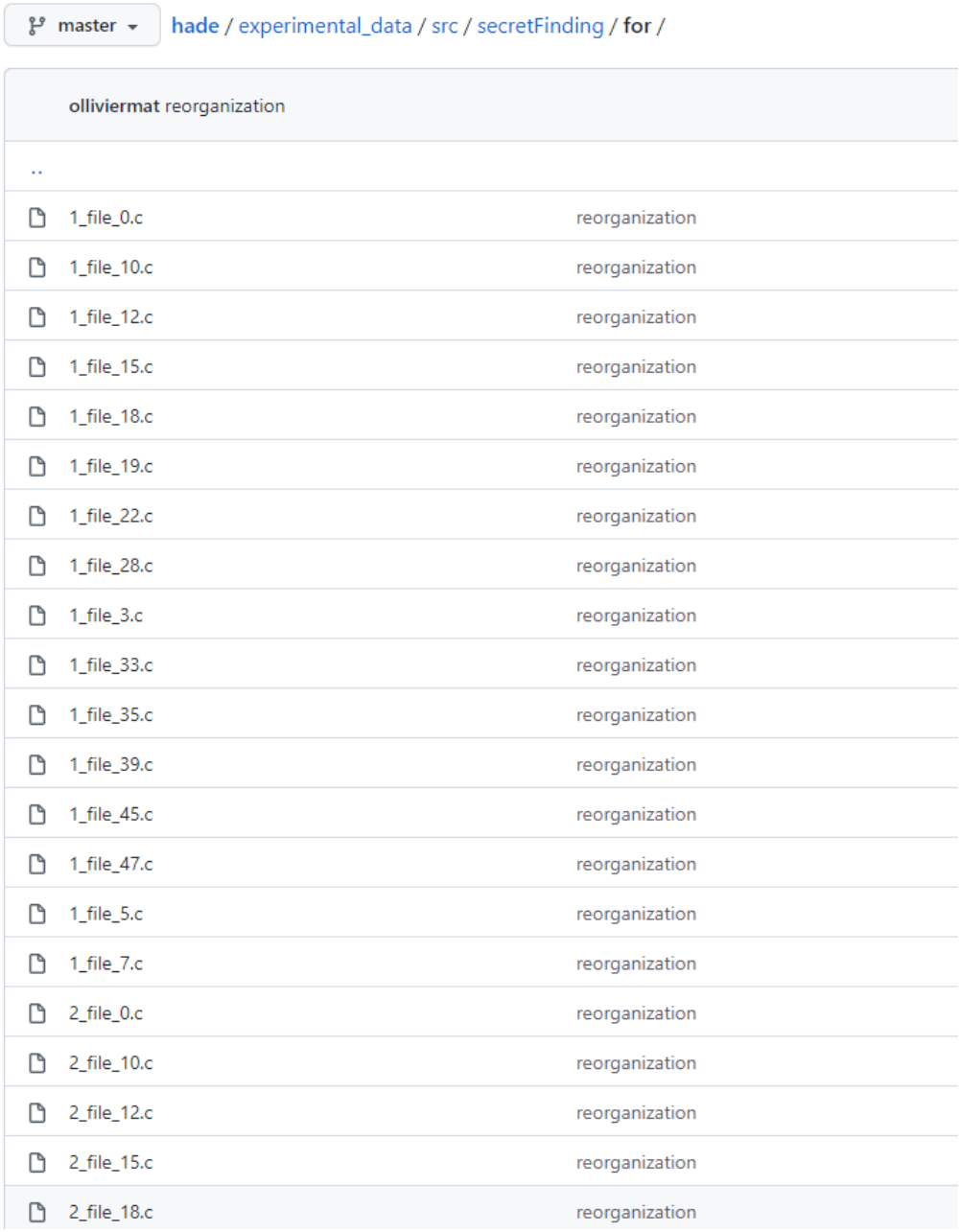
1 Use angr to deobfuscation for-based obfuscation

In the experiment, you use KLEE to attack 15 obfuscated programs, and when k = 5(four "for" loop was created), KLEE didn't attack any obfuscated program successfully in 3h.

Table 3: Impact of obfuscations on DSE

Transformation (#TO/#Samples)	Dataset #1		Dataset #2		
	Goal 1 3h TO	*Goal 2 1h TO*	*Goal 1 24h TO*	*Goal 2 3h TO*	*Goal 2 8h TO*
Virt	0/46	0/15	0/7	0/7	0/7
Virt ×2	1/46	0/15	0/7	0/7	0/7
Virt ×3	5/46	2/15	1/7	0/7	0/7
SPLIT ($k = 10$)	1/46	0/15	0/7	0/7	0/7
SPLIT ($k = 13$)	4/46	0/15	1/7	1/7	0/7
SPLIT ($k = 17$)	18/46	2/15	3/7	2/7	1/7
FOR ($k = 1$)	2/46	0/15	0/7	0/7	0/7
FOR ($k = 3$)	30/46	8/15	3/7	2/7	1/7
FOR ($k = 5$)	46/46	15/15	7/7	7/7	7/7

I download the 15 obfuscated programs from your github repo: https://github.com/binsec/hade. The prefix of the file name means the number of "for" loop.



I use Angr to attack these 15 obfuscated programs with Veritesting option open, and the results show that angr can attack them without much time cost. As shown in table1, it seems that Angr can attack these obfuscated programs without much time cost. I had uploaded the source code of my experiment in github: https://github.com/KiNa4Uc/questions

Obfuscated Program	k=3(with 3 ”for” loop)	k=4(with 4 ”for” loop)	Attack Successfully?
file_0	82s	141s	√
file_3	85s	137s	√
file_5	93s	147s	√
file_7	87s	140s	√
file_10	123s	198s	√
file_12	170s	268s	√
file_15	80s	131s	√
file_18	122s	194s	√
file_19	81s	141s	√
file_22	83s	131s	√
file_28	123s	212s	√
file_35	80s	130s	√
file_39	82s	131s	√
file_45	83s	131s	√
file_47	80s	131s	√

Table 1: How much time does Angr need to deal with for-based obfuscation

2 The obfuscation in figure 8

In fact, angr can deobfuscation all the loop-based obfuscation, I modify the 3_file_22.c:

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {

  int garb2 = 0;
  for (int it=0; it<argv[1][2]; it++) {
    garb2 = (garb2 ^ 1) + 2 * (garb2 & 1);
  }


  int garb1 = 0;
  for (int it=0; it<argv[1][1]; it++) {
    garb1 = (garb1 ^ 1) + 2 * (garb1 & 1);
  }


  int garb0 = 0;
  for (int it=0; it<argv[1][0]; it++) {
    garb0 = (garb0 ^ 1) + 2 * (garb0 & 1);
  }

  unsigned char c = argv[1][0];

  if (c > 127)
  printf("if-1-win ");
```

```
  else
  printf("if-1-lose ");

  if (c == 63)
  printf("if-2-win\n");
  else
  printf("if-2-lose\n");

  if ( garb2 == 127 ) {
    puts("Success !");
    //klee_report_error("err",16,"Secret found!","txt");
  }
  return 0;
}
```

And angr can attack it successfully in 120s.

```
2
WARNING | 2022-03-09 12:43:33,831 | angr.storage.memory_mixins.d
WARNING | 2022-03-09 12:43:33,834 | angr.storage.memory_mixins.d
WARNING | 2022-03-09 12:43:33,835 | angr.storage.memory_mixins.d
WARNING | 2022-03-09 12:43:33,836 | angr.storage.memory_mixins.d
WARNING | 2022-03-09 12:43:33,838 | angr.storage.memory_mixins.d
WARNING | 2022-03-09 12:43:33,840 | angr.storage.memory_mixins.d
SECRET: 0x4012a9
WARNING | 2022-03-09 12:43:34,307 | angr.storage.memory_mixins.d
Deprecation warning: Use self.model.get_any_node() instead of ge
WARNING | 2022-03-09 12:43:34,468 | angr.storage.memory_mixins.d
WARNING | 2022-03-09 12:43:49,737 | angr.sim_manager | Cannot fi
WARNING | 2022-03-09 12:44:14,319 | angr.sim_manager | Cannot fi
WARNING | 2022-03-09 12:44:17,152 | angr.storage.memory_mixins.d
WARNING | 2022-03-09 12:44:29,228 | angr.sim_manager | Cannot fi
WARNING | 2022-03-09 12:44:53,158 | angr.sim_manager | Cannot fi
WARNING | 2022-03-09 12:45:29,823 | angr.sim_manager | Cannot fi
WARNING | 2022-03-09 12:45:33,573 | angr.storage.memory_mixins.d
WARNING | 2022-03-09 12:45:34,259 | angr.storage.memory_mixins.d
WARNING | 2022-03-09 12:45:34,783 | angr.storage.memory_mixins.d
WARNING | 2022-03-09 12:45:35,447 | angr.storage.memory_mixins.d
a.out [121.96048188209534, '0x7f7f7f00']
```

Angr can not attack the recursive-based obfuscation successfully, because it can not find the merge point. However, I think if angr inline the recursive function 256 times, and use SSE to execute the generated code, then it can also deal with this type obfuscation.

3 My insight

The Z3-Solver is so powerful. I think that's why the Veritesting can deal with such obfuscation. Z Wang proposed a novel obfuscation based 3x+1 conjecture. Even though the loop times reach 95, z3 can output the result in 1s.

```
from z3 import *
```

```
2
3  a = BitVec("a", 32)
4  s = Solver()
5  for i in range(95):
6    a = If(a % 2 == 0, a / 2, a * 3 + 1)
7
8  s.add(a == 1)
9  s.check()
10 print(s.model())
```