```
 1 void _obstack_free (struct obstack *h, void *obj){
 2   struct _obstack_chunk *lp;
 3   struct _obstack_chunk *plp;
 4   lp = h->chunk;
 5   while (lp != 0 && ((void *) lp >= obj ||
 6             (void *) (lp)->limit < obj))
 7     { ...; }
 8   if (lp) { ...; }
 9   else if (obj != 0) { abort (); }
10 }
```

∗ The above is a bug-revealing example found in our benchmark (obstack.c in m4 library code), which is a part of a widely used memory allocation library in many software systems, such as GLibc [43], GNU binutils [4], and GNU Coreutils [18]. The *abort* function in Line 9 can be triggered by a non-null pointer obj, which is one of the parameters of the function _obstack_free. SymLoc enables the exploring of different values of obj, thus triggering the *abort*. The risk of the abort is that the parameters passed into the library code from client code may be wrong, thus causing system crashes and even DoS attacks [22]. Thus, SymLoc can help to effectively check proper usages of the memory management APIs in client code.

Figure 8: Implication 2 of pointer related bugs reported by SymLoc: boosting API usage checking

```
1 int main() {
2     char *buf = (char*) malloc(100 * sizeof(char));
3     free(buf+50); // an InF bug
4     return 0;
5 }
```

∗ The above tiny example shows how could SymLoc detect InF bugs faster. The freed pointer “buf+50” is not pointing to the beginning of the heap object buf, thus leading to an InF bug in Line 4. Although existing symbolic executors (e.g., KLEE) can detect the error, SymLoc is able to detect it in a more lightweight manner. Specifically, since the variable buf is symbolic, SymLoc simply checks the types of the symbolic expression to be freed. Here, the type of the symbolic expression “buf+50” is “Add” rather than “Concat”, meaning the pointer buf to be freed is not pointing to the beginning of the heap object, so SymLoc will report the InF bug. In contrast, existing executors (e.g., KLEE) send the concrete address of “buf+50” to the constraint solver to check whether it is inbound and at the beginning of the bound. Such a solution may intensify the program of constraint solving and performance in symbolic execution, while SymLoc can detect the bug without the heavy constraint solving.

Figure 9: A tiny example of InF bug

```
int main() {
    int *buff = (int *) malloc (100000 * sizeof(int));
    for (int i = 0; i < 100000; i++) {
        buff[i] = i; // memory write via a symbolic address
        printf(buff[i]); // read via a symbolic address
    }
    return 0;
}
```

∗ The above code simply allocates a buffer buff and iteratively writes a value and reads the value from 1 to 100000. We ran this code to compare the efficiency of SymLoc with RAM. RAM spent 55 seconds to finish the operations while SymLoc only takes 13 seconds. When the situation becomes complicated, e.g., more complex path constraints involving dynamically allocated addresses are involved, the performance downside may be amplified. Overall, SymLoc could be a complementary approach to RAM and other symbolic execution techniques, where RAM leverages more flexible symbolic addresses to support faster constraint solving, and SymLoc aims to facilitate a more comprehensive exploration of pointer-related paths.

Figure 10: Example code compared with RAM [67]

A APPENDIX: CODE EXAMPLES

```
 1 int main() {
 2     char temp;
 3     char *buff = (char *) malloc(100 * sizeof(char));
 4     klee_make_symbolic(&buff, sizeof(char*), "buff");
 5     if (buff > 0x54442343) {
 6         buff[1] = '9'; //write to a symbolic address
 7         ...; //code to be explored further
 8     } else {
 9         temp = buff[1]; //read from a symbolic address
10         ...;  //code to be explored further
11     }
12     return 0;
13 }
```

∗ Considering the above example code when KLEE is applied to explore the *if-else* branches, when we use the KLEE’s API klee_make_symbolic to make the pointer buff symbolic, KLEE can fork two execution states, but its exploration of the *if* and *else* branches will fail (due to invalid concretized addresses) or take a long time (because it tries to explore all possible concrete values for buff) when it encounters the write to the symbolic buff[1] (Line 6) or read from the symbolic address (Line 9). In contrast, SymLoc enables the read/write supported by the underlying concrete buffer associated with the symbolic buff, and thus can explore both of the *if/else* branches smoothly and potentially explore more execution paths.

Figure 6: Example code to illustrate the memory read/write via symbolic addresses in KLEE

```
1 char * buf = (char *) malloc (100);
2 if (buf == 0) // error-checking style 1
3     error();
```

```
1 if (buf !=0 ) { // error-checking style 2
2     buf[0] = '1';
3 }
4 ...//other code for exploration
5 buf[0] = '1';
```

∗ Some bugs reported by SymLoc only occur when the allocated address is *NULL* (i.e., 0). As shown in the code above, there are two common error-checking styles in the benchmark programs. The error in *Style* 1 is easy to explore by SymLoc and other static analysis tools (e.g., Frama-C [19]). However, if some code is written in *Style* 2, a programmer may mistakenly access a buffer (Line 5) outside of the checking code (Lines 1-3). It is too easy for other existing symbolic executors to overlook the implicit error condition at Line 5. In contrast, SymLoc is able to detect it as it easily forks two execution states for the buf!=0 condition to continue the following path exploration; then, one of the states will reveal the error that writes to an address of 0. In short, SymLoc is more capable of exploring pointer-related error-handling code.

Figure 7: Implication 1 of pointer related bugs reported by SymLoc: better checking of error-handling code