ESBMC memory model

Fedor Shmarov

ESBMC in a nutshell

- *“Do all **reachable assertions** in the program hold for all possible input values?”*

- “Is there an input value that leads to an assertion violation?”
  - **No?** The program is ***safe/correct***
  - **Yes?** Return the ***bug-trace/counter-example/violation witness***

Example

C program

```
1. int main {
2.   int x = __input();
3.   assert(x > 0);
4.   return 0;
5. }
```

Symbolic trace in SSA

```
int x_1 = __input();
assert(x_1 > 0);
```

SMT formula

$$\exists x_1 \in Dom(x_1): x_1 \leq 0$$

The verdict is: "one of the assertions does not hold"

line 2: x = 0
line 3: assert(0 > 0)

ESBMC memory model

- How are the objects created and destroyed?
- How are the objects accessed and modified?
- What is their relative location?

“ESBMC memory is a collection of typed symbols”

Creating and destroying objects (stack)

- Every time a new variable is declared, we create a new symbol

```
int x,y;
float a,b;
char p[5];
void *q;
```

- Symbols are never removed from “memory”
- They are internally marked as ***dead*** when their lifetime ends

```
…
int x;
dead(x);
return 0;
…
```

Symbol	Type
x	int
y	int
a	float
b	float
p	int[5]
q	void*

Creating and destroying objects (heap)

- Every time memory is allocated dynamically, we create a new byte array symbol

  int *ptr = malloc(42);

- All allocation sizes and validity are tracked through intrinsic data structure inside ESBMC

  __ESBMC_alloc_size[dyn_obj_1] = 42;

  __ESBMC_is_allocated[dyn_obj_1] = true;

- __ESBMC_is_allocated is updated when memory is freed

  __ESBMC_is_allocated[dyn_obj_1] = false;

Symbol	**Type**
ptr	int*
dyn_obj_1	char[]

Accessing and modifying objects (non-pointer)

- Every time a variable is updated, we create a new symbol
- Every time a variable is accessed via its most “recent” symbol

```
…
int x = __input();
x = x + 1;
assert(x > 0);
…
```

- This is better seen in SSA

```
…
int x_1 = __input();
x_2 = x_1 + 1;
assert(x_2 > 0);
…
```

Symbol	**Type**
x_1	int
x_2	int

Example with branches

C program

```c
int main {
  int x = __input();
  if(x >= 0) {
    x = x + 1;
  }
  assert(x > 0);
  return 0;
}
```

Symbolic trace in SSA

```
int x_1 = __input();
x_2 = x_1 + 1;
x_3 = phi(x_1 >= 0, x_2, x_1);
assert(x_3 > 0);
```

SMT formula

$$\exists x_1 \in Dom(x_1): (x_2 = x_1 + 1) \land$$
$$\big((x_1 \geq 0) \rightarrow (x_3 = x_2)\big) \land$$
$$\big((x_1 < 0) \rightarrow (x_3 = x_1)\big) \land$$
$$(x_3 \leq 0)$$

Accessing and modifying objects (pointers)

- Pointer accesses are done via a pair *{obj, offset}*
- Every time a pointer is dereferenced, we consider every possible access to every object with the corresponding offset
- All memory safety properties are encoded as assertions

```
…
int x = __input();
int *ptr;
if(x>0) {
  ptr = malloc(40);
} else {
  ptr = malloc(20);
}
ptr[5] = 0;
…
```

Accessing and modifying objects (pointers)

```
…
if(x_1>0) {
  // int *ptr = malloc(40);
  char dyn_obj_1[40];
  __ESBMC_alloc_size[dyn_obj_1] = 40;
  __ESBMC_is_allocated[dyn_obj_1] = true;
} else {
  // int *ptr = malloc(20);
  char dyn_obj_2[20];
  __ESBMC_alloc_size[dyn_obj_2] = 20;
  __ESBMC_is_allocated[dyn_obj_2] = true;
}
dyn_obj_3 = phi(x_1 > 0, dyn_obj_1, dyn_obj_2);
// ptr[5] = 0;
assert(20 >= 0 && 20 + 4 <= __ESBMC_alloc_size[dyn_obj_3]);
assert(__ESBMC_is_allocated[dyn_obj_3]);
dyn_obj_3[20] = __extract_byte(0,0);
dyn_obj_3[21] = __extract_byte(1,0);
dyn_obj_3[22] = __extract_byte(2,0);
dyn_obj_3[23] = __extract_byte(3,0);
…
```

Additional assumptions

- Objects are completely disjoint from each other
  - It is impossible to “jump” between objects via pointer arithmetics
  - This is OK because we do not allow object bounds violations
- Objects can be allocated into previously freed memory
- Extra safety checks are encoded as assertions
  - Alignment requirements
  - Arithmetic overflows
  - Capability validity
  - …

Thank you