SE547: Lecture 8

Overview

Software Security

Software Model Checking

Using CBMC

CBMC Algorithm

CBMC Features

Software Security

Recall...

What is a buffer overflow attack?

What is the result of a successful buffer overflow attack?

What can we do about buffer overflows?

What is a SAT-solver?

What is a software model checker?

Software Model Checking

Goal of a software model checker: to ensure a program is *safe*. What does this mean?

What are example safety properties we might expect for C programs?

Imagine we have a function `boolean safe(String filename)` which decides safety: how can we reach a contradiction?

This means that verifying safety is undecidable. What can we do about it?

Common solution to undecidability: *bounded* model checkers.

Using CBMC

Consider mkstring-buggy.c:

```
char* mkstring (char c, int size) {
  char* result = NULL;
  int i=0;
  result = (char*)(malloc((size+1)*sizeof(char)));
  for (i=0; i<size; i++) {
    result[i]=c;
  }
  result[size]=0;
  return result;
}
```

What is this program meant to do? What is wrong with it?

Using CBMC

Download cbmc-ui-2-0-win.zip and ansi-c-lib.zip

```
cbmc --decide --unwind 5 --no-unwinding-assertions --function mkstring mkstring-buggy.c
```

What those options mean:

cbmc --decide *run the model checker*
- --unwind 5 *expand the for-loop 5 times*
- --no-unwinding-assertions *don't issue warnings about --unwind*
- --function mkstring *which function to check*
- mkstring-buggy.c *filename*

What edits do we need to make to get this program to pass the test?

CBMC Algorithm

How cbmc works:

1) Start with a C program with assert(...) and assume(...) statements.

2) Preprocess the program: just left with functions, loops, if, goto and assignment.

3) Unwind function calls, loops and goto: just left with if and assignment.

4) Rename all variables to be unique.

5) Calculate the *constraints* of the program C, and the *properties* we're checking of the program P.

6) Hand off checking $C \Rightarrow P$ to a SAT-solver.

Hooray, we've translated an undecidable problem into an NP-complete problem: where has all the work gone?

CBMC Algorithm

1) Start with a C program with assert(...) and assume(...) statements.

```
void test (int x, int y) {
  int i; int z=0;
  assume (x < y);
  for (i=x; i<y; i++) { z++; }
  assert (z > 0);
}
```

assume-assert.c Is this program OK?

CBMC Algorithm

2) Preprocess the program: just left with functions, loops, if, goto and assignment.

```
void test (int x, int y) {
  int i; int z; z=0;
  assume (x < y);
  i=x;
  while (i<y) { z=z+1; i=i+1; }
  assert (z > 0);
}
```

CBMC Algorithm

3) Unwind function calls and loops: just left with if, forward goto and assignment.

```
void test (int x, int y) {
  int i; int z; z=0;
  assume (x < y);
  i=x;
  if (i<y) { z=z+1; i=i+1; }
  if (i<y) { z=z+1; i=i+1; }
  if (i<y) { z=z+1; i=i+1; }
  if (i<y) { assume (false); } // what is this doing here?
  assert (z > 0);
}
```

This is for –unwind 3 –no-unwinding-assertions.

CBMC Algorithm

4) Rename all variables to be unique.

```
void test (int x0, int y0) {
  int i0; int z0; int i1; int z1; ... int i6; int z6; z0=0;
  assume (x0 < y0);
  i0=x0;
  if (i0<y0) { z1=z0+1; i1=i0+1; }
  z2=(i0<y0?z1,z0); i2=(i0<y0?i1,i0); // what is this doing here?
  if (i2<y0) { z3=z2+1; i3=i2+1; }
  z4=(i2<y0?z3,z32; i4=(i2<y0?i3,i2); // what is this doing here?
  if (i4<y0) { z5=z4+1; i5=i4+1; }
  z6=(i4<y0?z5,z4); i6=(i4<y0?i5,i4); // what is this doing here?
  if (i6<y0) { assume (false); }
  assert (z6 > 0);
}
```

(If you've taken CSC548: this is SSA form with phi-functions.)

CBMC Algorithm

5) Calculate the *constraints* of the program *C*, and the *properties* we're checking of the program *P*.

Constraints:

```
z0=0;
x0 < y0;
i0=x0;
i0<y0 => z1=z0+1;  i0<y0 => i1=i0+1;
z2=(i0<y0?z1,z0); i2=(i0<y0?i1,i0);
i2<y0 => z3=z2+1;  i2<y0 => i3=i0+1;
z4=(i2<y0?z3,z2); i4=(i2<y0?i3,i2);
i4<y0 => z5=z4+1;  i4<y0 => i5=i0+1;
z6=(i4<y0?z5,z4); i6=(i4<y0?i5,i4);
i6<y0 => false
```

Properties:

```
z6 > 0;
```

CBMC Algorithm

6) Hand off checking $C \Rightarrow P$ to a SAT-solver.

Solving with ZChaff version ZChaff 2003.6.16
1222 variables, 3948 clauses
SAT checker: negated claim is UNSATISFIABLE, i.e., holds
VERIFICATION SUCCESSFUL

We talked about SAT-solvers last week.

CBMC Algorithm

Do an example...

```
int factorial (int x) {
  int result;
  if (x < 2) { result = 1; } else { result = factorial (x-1) * x; }
  assert (result > 0);
}
```

factorial.c Is this program OK?

CBMC Algorithm

The tricky part is:

5) Calculate the *constraints* of the program C, and the *properties* we're checking of the program P.

For a program I, the constraints are $C(I,\text{true})$ where $C(I,g)$ is defined:

$C(I;J, g) = C(I, g) \wedge C(I, g)$
$C(\text{if } (c) \{ I \} \text{ else } \{ J \}, g) = C(I, g \wedge c) \wedge C(J, g \wedge \neg c)$
$C(\text{assume}(c), g) = g \Rightarrow c$
$C(x{=}e, g) = g \Rightarrow x{=}e$
$C(\text{anything else}, g) = \text{true}$

For a program I, the properties are $P(I,\text{true})$ where $P(I,g)$ is defined:

$P(I;J, g) = P(I, g) \wedge P(I, g)$
$P(\text{if } (c) \{ I \} \text{ else } \{ J \}, g) = P(I, g \wedge c) \wedge P(J, g \wedge \neg c)$
$P(\text{assert}(c), g) = g \Rightarrow c$
$P(\text{anything else}, g) = \text{true}$

CBMC Features

CBMC handles integer and fixed point arithmetic.

Checks for: overflow, division by zero.

CBMC Features

CBMC handles arrays and memory management, for example:

```
void sort (int* array, int size) {
  int tmp; int i; int j;
  for (i=0; i<size; i++) {
    for (j=i; j<size; j++) {
      if (array[i] > array[j]) {
        tmp = array[i]; array[i] = array[j]; array[j] = tmp;
      }
    }
  }
}

void test (int size) {
  int* array = (int*)(malloc(size*sizeof(int)));
  sort (array, size);
}
```

Includes tests for array bounds! sort.c

Summary

CBMC is a C Bounded Model Checker. It can help with bug hunting by exhaustively searching for failed assertions.

CBMC translates C down to predicate logic, and throws the problem to a SAT-solver.

The resulting system does not catch every bug, but it catches quite a lot!

Next week

Proof-carrying code.