Bounded Software Model Checking with CBMC

Jan Tobias Mühlberg

DistriNet, Dept. of Computer Science, K.U.Leuven, Belgium,
jantobias.muehlberg@cs.kuleuven.be

Leuven, 18th November 2011

Outline

Session I: 18^{th} November 2011

- Motivating examples
- Finding Security Vulnerabilities with CBMC
- Introduction to the Assignment
- Some homework tasks

Session II: 2^{nd} December 2011

- Discuss homework
- CBMC Background
- Q&A Session on CBMC and the assignment

Session I: 18^{th} November 2011

Session I

CBMC: Coordinates

The **C B**ounded **M**odel **C**hecker

- `http://www.cprover.org/`
- Originally by Clarke, Kroening and Lerda [CKL04]
- Currently maintained by Daniel Kroening at Oxford
- One of the key tools in PINCETTE
  `http://www.pincette-project.eu/`
- Supports very large subset of C and some C++
  `http://www.cprover.org/cbmc/language_features.html`

CBMC: Coordinates

The **C** **B**ounded **M**odel **C**hecker

- `http://www.cprover.org/`
- Originally by Clarke, Kroening and Lerda [CKL04]
- Currently maintained by Daniel Kroening at Oxford
- One of the key tools in PINCETTE
  `http://www.pincette-project.eu/`
- Supports very large subset of C and some C++
  `http://www.cprover.org/cbmc/language_features.html`
- There is also SatAbs, a checker based on predicate abstraction [CKSY05]

CBMC: Getting the Tool

Installation instructions

- Windows, Linux and MacOS packages are available at `http://www.cprover.org/cbmc/`
- For Windows you need Visual Studio or the free Visual C++ 2010 Express; we recommend using Linux or MacOS
- There is Eclipse support (`http://www.cprover.org/eclipse-plugin/`). I had some trouble installing it – do not waste too much time here.
- **All assignment tasks can be accomplished by using the command-line CBMC!**

Motivating examples

Tool Demo!

The demo is based on `course_01.c`, which is available on Toledo.

Finding Security Vulnerabilities with CBMC

Vulnerabilities (or programming errors that may lead to vulnerabilities) that you can find with CBMC:

- Pointer safety
- Buffer overflows
- Arithmetic overflow
- Generic user-defined assertions API safety rules

Finding Security Vulnerabilities with CBMC

Vulnerabilities (or programming errors that may lead to vulnerabilities) that you can find with CBMC:

- Pointer safety
- Buffer overflows
- Arithmetic overflow
- Generic user-defined assertions API safety rules

Bear in mind: CBMC performs a *bounded* analysis!

Introduction to the Assignment

This year's **assignment** in "Ontwikkeling van Veilige Software":

Introduction to the Assignment

This year's **assignment** in "Ontwikkeling van Veilige Software":

- *Finding security vulnerabilities with CBMC*
- Work in groups of 2 or 3 people, 30h per student
- Submit results by the **23rd of December 2011**

Introduction to the Assignment

This year's **assignment** in "Ontwikkeling van Veilige Software":

- *Finding security vulnerabilities with CBMC*
- Work in groups of 2 or 3 people, 30h per student
- Submit results by the **23rd of December 2011**

- You will **get a C program** with several vulnerabilities; you have to systematically **apply CBMC** to identify these vulnerabilities and to fix or even exploit them
- **Goals:** Understand bounded model checking; understand the vulnerabilities

Introduction to the Assignment

The Program:

- `addrbook.c`: a trivial address book implementation
- 350 LOC; a doubly-linked list of address entries; textual interface for managing entries

Introduction to the Assignment

The Program:

- `addrbook.c`: a trivial address book implementation
- 350 LOC; a doubly-linked list of address entries; textual interface for managing entries
- Intended to be used by other programs that implement the user interface
- Intended to run on an embedded device: OS takes care of persistently storing the program state

Introduction to the Assignment

The Program:

- `addrbook.c`: a trivial address book implementation
- 350 LOC; a doubly-linked list of address entries; textual interface for managing entries
- Intended to be used by other programs that implement the user interface
- Intended to run on an embedded device: OS takes care of persistently storing the program state
- Requirements: dependability and security

Introduction to the Assignment

The Mandatory Tasks:

- **Task 1:** Check that the program does not exhibit undefined behaviour when memory allocation fails.

Introduction to the Assignment

The Mandatory Tasks:

- **Task 1:** Check that the program does not exhibit undefined behaviour when memory allocation fails.
- **Task 2:** Implement a stack and check it for memory safety; experiment with loop bounds and unwinding assertions.

Introduction to the Assignment

The Mandatory Tasks:

- **Task 1:** Check that the program does not exhibit undefined behaviour when memory allocation fails.
- **Task 2:** Implement a stack and check it for memory safety; experiment with loop bounds and unwinding assertions.
- **Task 3:** Extract the list-implementation from the address book and verify it for memory safety.

Introduction to the Assignment

The Mandatory Tasks:

- **Task 1:** Check that the program does not exhibit undefined behaviour when memory allocation fails.
- **Task 2:** Implement a stack and check it for memory safety; experiment with loop bounds and unwinding assertions.
- **Task 3:** Extract the list-implementation from the address book and verify it for memory safety.
- **Task 4:** Verify the address book.

Introduction to the Assignment

The Mandatory Tasks:

- **Task 1:** Check that the program does not exhibit undefined behaviour when memory allocation fails.
- **Task 2:** Implement a stack and check it for memory safety; experiment with loop bounds and unwinding assertions.
- **Task 3:** Extract the list-implementation from the address book and verify it for memory safety.
- **Task 4:** Verify the address book.
- **Task 5:** Discuss how you would write a program that is to be verified with CBMC (given the limitations of bounded model checking and CBMC that you have explored).

Introduction to the Assignment

The Optional Tasks:

- **Task 6:** What are format string vulnerabilities and how can CBMC be used to identify them in a given program?

Introduction to the Assignment

The Optional Tasks:

- **Task 6:** What are format string vulnerabilities and how can CBMC be used to identify them in a given program?
- **Task 7:** Check the address book for format string vulnerabilities.

Introduction to the Assignment

The Optional Tasks:

- **Task 6:** What are format string vulnerabilities and how can CBMC be used to identify them in a given program?
- **Task 7:** Check the address book for format string vulnerabilities.
- **Task 8:** If you find any format string vulnerabilities, exploit them.

Introduction to the Assignment

The Optional Tasks:

- **Task 6:** What are format string vulnerabilities and how can CBMC be used to identify them in a given program?
- **Task 7:** Check the address book for format string vulnerabilities.
- **Task 8:** If you find any format string vulnerabilities, exploit them.
- **Task 9:** Compare the CBMC approach with testing techniques or other formal methods you know.

Introduction to the Assignment

The Optional Tasks:

- **Task 6:** What are format string vulnerabilities and how can CBMC be used to identify them in a given program?
- **Task 7:** Check the address book for format string vulnerabilities.
- **Task 8:** If you find any format string vulnerabilities, exploit them.
- **Task 9:** Compare the CBMC approach with testing techniques or other formal methods you know.

Good luck!

Getting started

Where to start

- Do the **homework** task.
- CBMC website has tutorials and documentation
- CBMC manual:
  `http://www.cprover.org/cbmc/doc/manual.pdf`
- Read the assignment text!
- Write little toy examples to isolate problems.

Getting started

Where to start

- Do the **homework** task.
- CBMC website has tutorials and documentation
- CBMC manual:
  `http://www.cprover.org/cbmc/doc/manual.pdf`
- Read the assignment text!
- Write little toy examples to isolate problems.

If you get stuck

- There will be a *Q&A Session* on the **2nd of December**.
- Use the course forums on *Toledo* for further questions. For technical questions always add **small** example code that illustrates the problem.

Homework

Verify Bubblesort

Homework

Verify Bubblesort

- `bubble.c` contains an implementation of the Bubblesort algorithm

Homework

Verify Bubblesort

- `bubble.c` contains an implementation of the Bubblesort algorithm
- **1:** Add a function `main()` so as to allocate an array of integers, initialise the array elements and pass it to `bubblesort()`. Use CBMC to verify the program for pointer safety and the absence of array bounds overflows.

Homework

Verify Bubblesort

- `bubble.c` contains an implementation of the Bubblesort algorithm
- **1:** Add a function `main()` so as to allocate an array of integers, initialise the array elements and pass it to `bubblesort()`. Use CBMC to verify the program for pointer safety and the absence of array bounds overflows.
- **2:** Implement a function `sorted(int *array, int length)` which tests to ensure that an array is sorted. Use CBMC to verify that `assert (sorted (array, length))` holds.

Homework

Verify Bubblesort

- `bubble.c` contains an implementation of the Bubblesort algorithm
- **1:** Add a function `main()` so as to allocate an array of integers, initialise the array elements and pass it to `bubblesort()`. Use CBMC to verify the program for pointer safety and the absence of array bounds overflows.
- **2:** Implement a function `sorted(int *array, int length)` which tests to ensure that an array is sorted. Use CBMC to verify that `assert (sorted (array, length))` holds.
- **3:** Repeat for Quicksort.

Homework

Verify Bubblesort

```
void bubblesort (int *array, int length)
 {
  int i, j;

  for (i = 0; i < length - 1; ++i)
   { for (j = 0; j < length - i - 1; ++j)
      { if (array[j] > array[j + 1])
         { int tmp = array[j];

           array[j] = array[j + 1];
           array[j + 1] = tmp; }
      } }
 }
```

Homework

Verify Bubblesort

```
#include "bubble.c"

int sorted (int *array, int length)
 { /* ... check */ }

#define SIZE ...

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

   /* ... array initialisation */

   bubblesort (array, SIZE);
   assert (sorted (array, SIZE));

   return (0); }
```

Thank you!

Thank you! Questions?

Session II: 2nd December 2011

Session II

Homework

Questions and Problems?

Homework

Some issues you may have encountered:

- Array handling: copying bytes from one array (or `malloc`ed chunk) to another
- Multiple dereferences in one statement: `x->y->z = 5`
- Loop bounds and unwinding assertions
- Recursion
- Combinatorial blow-up: CBMC terminates with “out of memory”

Bounded Software Model Checking (as in CBMC)

CBMC is based on a technique described in by Biere et al. as "symbolic model checking without BDDs" [BCCZ99]:

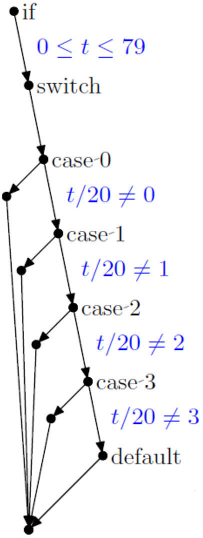
- Transform the program into a *control flow graph*
- Follow paths through the CFG to an assertion, and build a formula that corresponds to the path
- Formulae are instances of SAT or SMT: pass them to a solver and obtain a satisfying assignment
- ...repeat for all paths and assertions
- Satisfying assignments can be reused as input to the program

Bounded Software Model Checking (as in CBMC)

```
if ( (0 <= t) && (t <= 79) )
switch ( t / 20 )
{ case 0:
    TEMP2 = ( (B AND C) OR (~B AND D) );
    TEMP3 = ( K_1 );
    break;
  case 1:
    TEMP2 = ( (B XOR C XOR D) );
    TEMP3 = ( K_2 );
    break;
  case 2:
    TEMP2 = ( (B AND C) OR (B AND D) OR (C AND D) );
    TEMP3 = ( K_3 );
    break;
  case 3:
    TEMP2 = ( B XOR C XOR D );
    TEMP3 = ( K_4 );
    break;
  default:
    assert(0); }
```

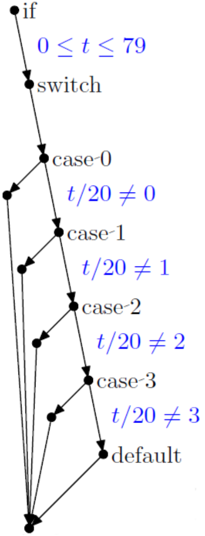
Bounded Software Model Checking (as in CBMC)

```
if ( (0 <= t) && (t <= 79) )
switch ( t / 20 )
{ case 0:
    TEMP2 = ( (B AND C) OR (~B AND D) );
    TEMP3 = ( K_1 );
    break;
  case 1:
    TEMP2 = ( (B XOR C XOR D) );
    TEMP3 = ( K_2 );
    break;
  case 2:
    TEMP2 = ( (B AND C) OR (B AND D) OR (C AND D)
    TEMP3 = ( K_3 );
    break;
  case 3:
    TEMP2 = ( B XOR C XOR D );
    TEMP3 = ( K_4 );
    break;
  default:
    assert(0); }
```



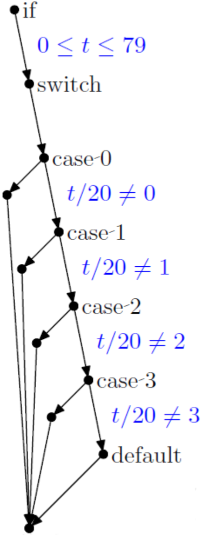
Bounded Software Model Checking (as in CBMC)

CFG

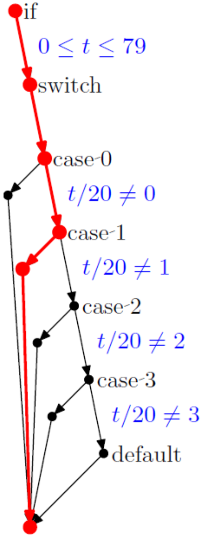


Bounded Software Model Checking (as in CBMC)

CFG

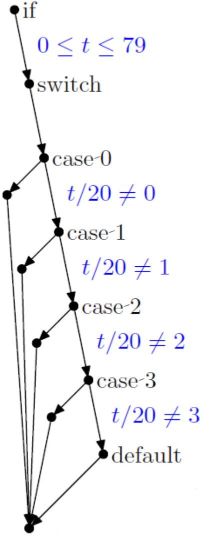


A Path

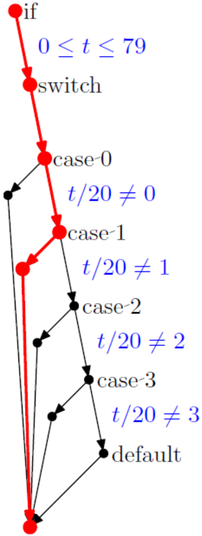


Bounded Software Model Checking (as in CBMC)

CFG



A Path

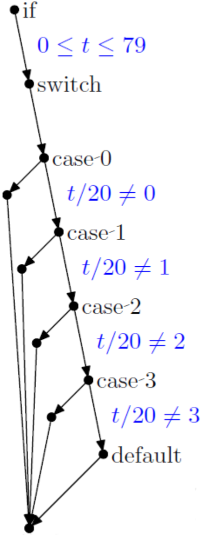


Formula

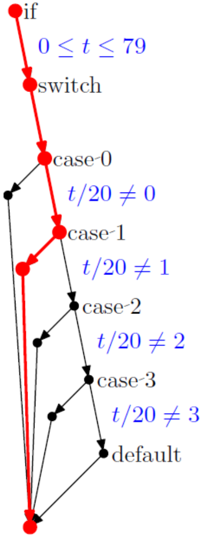
$$
\begin{aligned}
& 0 \leq \mathtt{t} \leq 79 \\
\wedge\ & \mathtt{t/20} \neq 0 \\
\wedge\ & \mathtt{t/20} = 1 \\
\wedge\ & \mathtt{TEMP2} = \mathtt{B} \oplus \mathtt{C} \oplus \mathtt{D} \\
\wedge\ & \mathtt{TEMP3} = \mathtt{K\_2}
\end{aligned}
$$

Bounded Software Model Checking (as in CBMC)

CFG



A Path



Formula

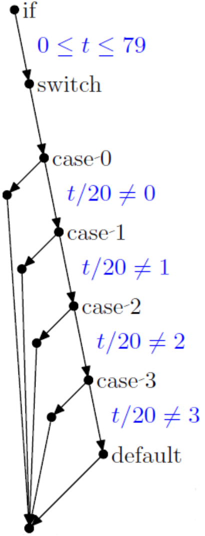
$0 \leq t \leq 79$
$\wedge$ t/20 $\neq$ 0
$\wedge$ t/20 = 1
$\wedge$ TEMP2 = B $\oplus$ C $\oplus$ D
$\wedge$ TEMP3 = K_2

Sat. Assign.

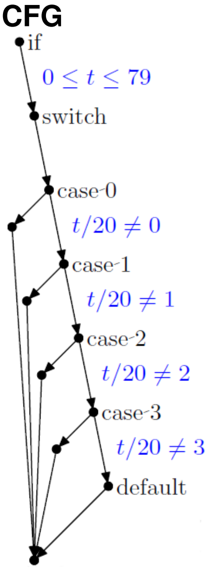
t $\mapsto$ 21; B $\mapsto$ 0;
C $\mapsto$ 0; D $\mapsto$ 0;
K_2 $\mapsto$ 10;
TEMP2 $\mapsto$ 0;
TEMP3 $\mapsto$ 10

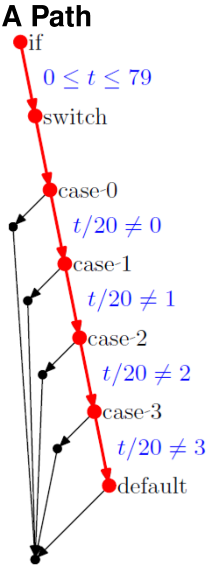
Bounded Software Model Checking (as in CBMC)

CFG



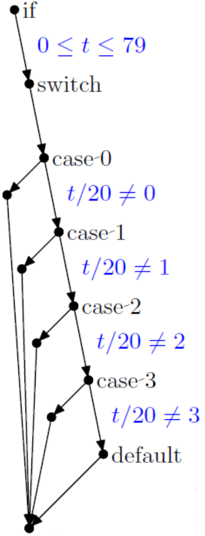
Bounded Software Model Checking (as in CBMC)



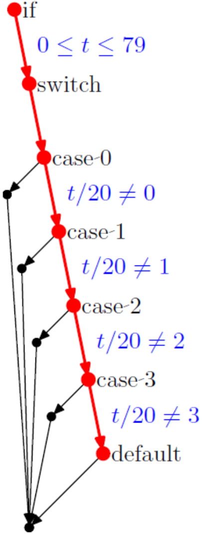


Bounded Software Model Checking (as in CBMC)

CFG



A Path

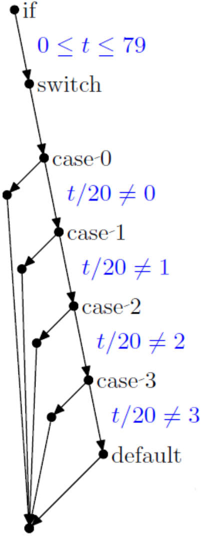


Formula

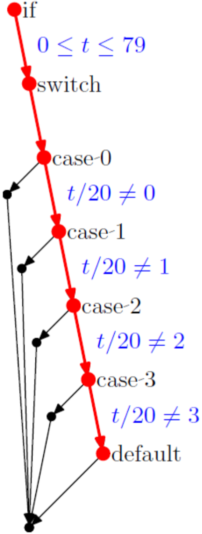
$$
\begin{array}{l}
\quad 0 \leq t \leq 79 \\
\wedge\ t/20 \neq 0 \\
\wedge\ t/20 \neq 1 \\
\wedge\ t/20 \neq 2 \\
\wedge\ t/20 \neq 3
\end{array}
$$

Bounded Software Model Checking (as in CBMC)

CFG



A Path



Formula

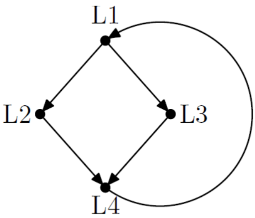
$$
\begin{aligned}
& 0 \leq t \leq 79 \\
\wedge\ & t/20 \neq 0 \\
\wedge\ & t/20 \neq 1 \\
\wedge\ & t/20 \neq 2 \\
\wedge\ & t/20 \neq 3
\end{aligned}
$$

Sat. Assign.

UNSAT

Bounded Software Model Checking (as in CBMC)

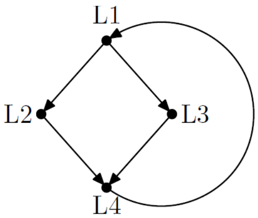
Let's consider the following CFG:



CBMC unwinds loops a bounded number of times.

Bounded Software Model Checking (as in CBMC)

Let's consider the following CFG:



CBMC unwinds loops a bounded number of times.

How many paths do we get for *n* iterations?

Bounded Software Model Checking (as in CBMC)

Tool Demo!

Summary

- Bounded Model Checking (BMC) is the most successful formal validation technique in the hardware industry

Summary

- Bounded Model Checking (BMC) is the most successful formal validation technique in the hardware industry
- Enabling technology: efficient decision procedures

Summary

- Bounded Model Checking (BMC) is the most successful formal validation technique in the hardware industry
- Enabling technology: efficient decision procedures
- Advantages:
  - Strongly automated
  - Robust
  - Finds subtle bugs

Summary

- Bounded Model Checking (BMC) is the most successful formal validation technique in the hardware industry
- Enabling technology: efficient decision procedures
- Advantages:
  - Strongly automated
  - Robust
  - Finds subtle bugs
- Great if: you only look for bugs up to specific depth or your programs are relatively “shallow”

Summary

- Bounded Model Checking (BMC) is the most successful formal validation technique in the hardware industry
- Enabling technology: efficient decision procedures
- Advantages:
  - Strongly automated
  - Robust
  - Finds subtle bugs
- Great if: you only look for bugs up to specific depth or your programs are relatively “shallow”
- Good for many applications, e.g., embedded systems

Thank you!

Thank you! Questions?

This talk is largely based on material from “CBMC: Bounded Model Checking for ANSI-C” available at

`www.cprover.org/cbmc/doc/cbmc-slides.pdf`

References I

A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu.
Symbolic model checking without bdds.
In *TACAS '99*, vol. 1579 of *LNCS*, pp. 193–207, Heidelberg, 1999. Springer.

E. Clarke, D. Kroening, and F. Lerda.
A tool for checking ANSI-C programs.
In *TACAS '04*, vol. 2988 of *LNCS*, pp. 168–176, Heidelberg, 2004. Springer.

E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav.
SATABS: SAT-based predicate abstraction for ANSI-C.
In *TACAS '05*, vol. 3440 of *LNCS*, pp. 570–574, Heidelberg, 2005. Springer.