Systems and Software Verification Laboratory

The University of Manchester

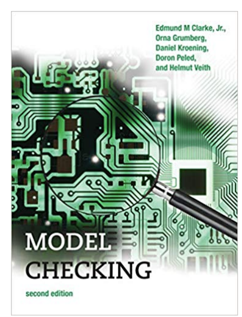
Detection of Software Vulnerabilities: Static Analysis

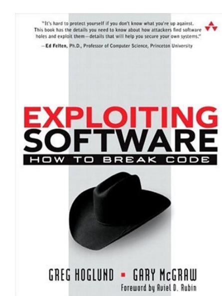
Lucas Cordeiro
Department of Computer Science
lucas.cordeiro@manchester.ac.uk

Detection of Software Vulnerabilities

- Lucas Cordeiro (Formal Methods Group)
  - *lucas.cordeiro@manchester.ac.uk*
  - Office: 2.28
  - Office hours: 15-16 Tuesday, 14-15 Wednesday
- Textbook:
  - *Model checking* (Chapter 14)
  - *Exploiting Software: How to Break Code* (Chapter 7)
  - *C How to Program* (Chapter 1)

Rashid et al.: *The Cyber Security Body of Knowledge*, CyBOK, v1.0, 2019





Intended learning outcomes

- Understand **soundness** and **completeness** concerning **detection techniques**
- Emphasize the difference between **static analysis** and **testing / simulation**
- Explain **Bounded** and **Unbounded Model Checking**
- Provide **practical examples** to detect software vulnerabilities statically

Intended learning outcomes

- Understand **soundness** and **completeness** concerning **detection techniques**
- Emphasize the difference between **static analysis** and **testing / simulation**
- Explain **Bounded** and **Unbounded Model Checking**
- Provide **practical examples** to detect software vulnerabilities statically

Motivating Example

- functionality demanded increased significantly
  - peer reviewing and testing
- multi-core processors with scalable shared memory / message passing
  - software model checking and testing

```
void *threadA(void *arg) {
  lock(&mutex);
  x++;
  if (x == 1) lock(&lock);
  unlock(&mutex); (CS1)
  lock(&mutex);   (CS3)
  x--;
  if (x == 0) unlock(&lock);
  unlock(&mutex);
}
```

Deadlock

```
void *threadB(void *arg) {
  lock(&mutex);
  y++;
  if (y == 1) lock(&lock); (CS2)
  unlock(&mutex);
  lock(&mutex);
  y--;
  if (y == 0) unlock(&lock);
  unlock(&mutex);
}
```

Detection of Vulnerabilities

- Detect the presence of vulnerabilities in the code during the development, testing and maintenance
- Techniques to detect vulnerabilities must make trade-offs between **soundness** and **completeness**
  - A detection technique is **sound** for a given category if it concludes that a given program has no vulnerabilities
    - An unsound detection technique may have ***false negatives***, i.e., actual vulnerabilities that the detection technique fails to find
  - A detection technique is **complete** for a given category, if any vulnerability it finds is an actual vulnerability
    - An incomplete detection technique may have ***false positives***, i.e. it may detect issues that do not turn out to be actual vulnerabilities

Detection of Vulnerabilities

- Achieving **soundness** requires reasoning about all executions of a program (usually an infinite number)
  - This is can done by static checking of the program code while making suitable abstractions of the executions
- Achieving **completeness** can be done by performing actual, concrete executions of a program that are witnesses to any vulnerability reported
  - The analysis technique has to come up with concrete inputs for the program that trigger a vulnerability
    - A common dynamic approach is software testing: the tester writes test cases with concrete inputs, and specific checks for the outputs

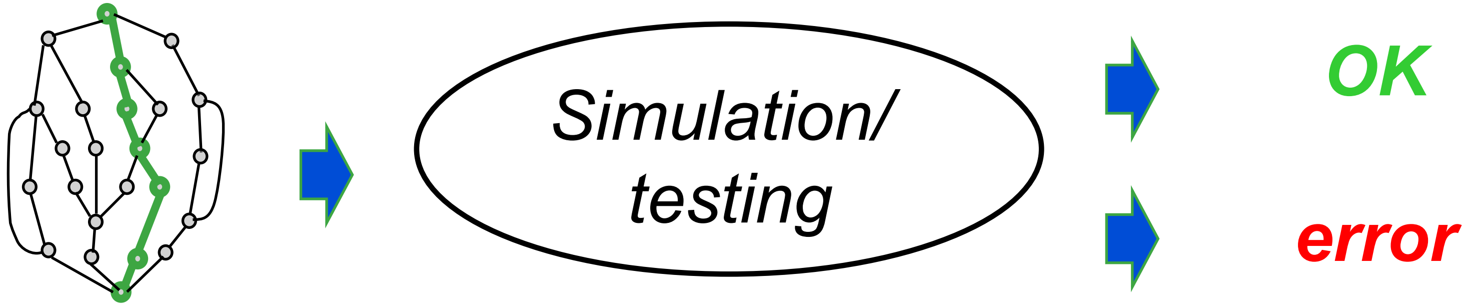
Detection of Vulnerabilities

In practice, detection tools can use a **hybrid combination of static and dynamic analysis** techniques to achieve a good trade-off between **soundness and completeness**

Intended learning outcomes

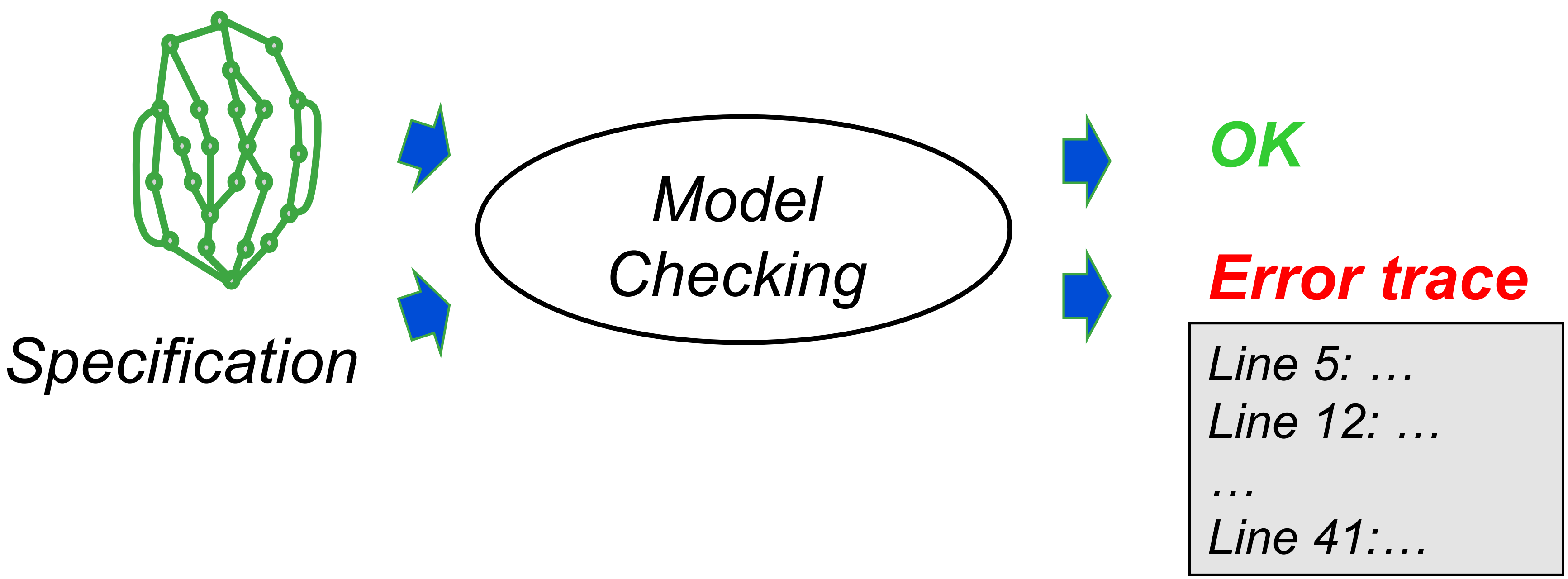
- Understand **soundness** and **completeness** concerning **detection techniques**
- Emphasize the difference between **static analysis** and **testing / simulation**
- Explain **Bounded** and **Unbounded Model Checking**
- Provide **practical examples** to detect software vulnerabilities statically

Static analysis vs Testing/ Simulation



- Checks only some of the system executions
- May miss errors

Static analysis vs Testing/ Simulation

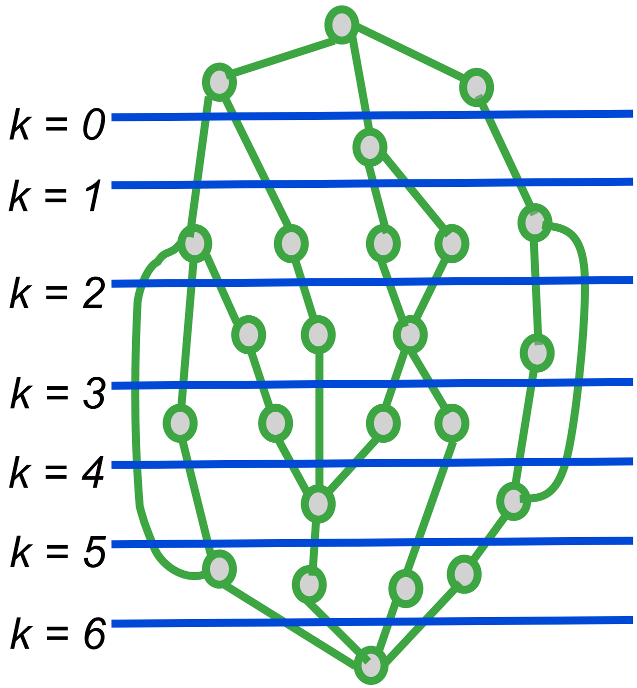


- Exhaustively explores all executions
- Report errors as traces

Avoiding state space explosion

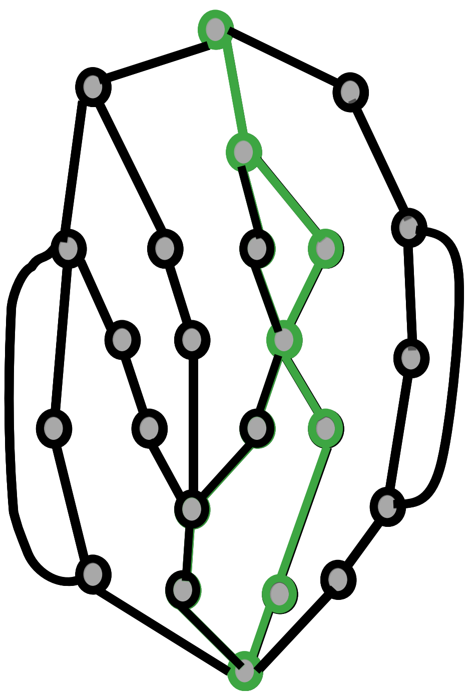
- Bounded Model Checking (BMC)
  - Breadth-first search (BFS) approach
- Symbolic Execution
  - Depth-first search (DFS) approach

Bounded Model Checking



- Bounded model checkers explore the state space in depth
- Can only prove correctness if all states are reachable within the bound

Symbolic Execution



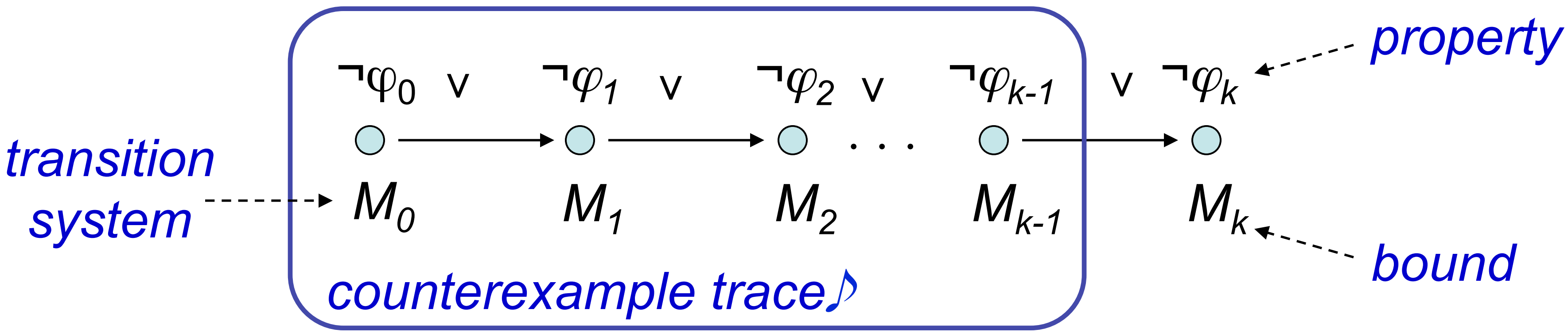
- Symbolic execution explores all paths individually
- Can only prove correctness if all paths are explored

Intended learning outcomes

- Understand **soundness** and **completeness** concerning **detection techniques**
- Emphasize the difference between **static analysis** and **testing / simulation**
- Explain **Bounded** and **Unbounded Model Checking**
- Provide **practical examples** to detect software vulnerabilities statically

Bounded Model Checking

Basic Idea: check negation of given property up to given depth



- transition system M unrolled k times
  - for programs: unroll loops, unfold arrays, …
- translated into verification condition ψ such that

  ψ satisfiable iff φ has counterexample of max. depth k
- has been applied successfully to verify (sequential) software

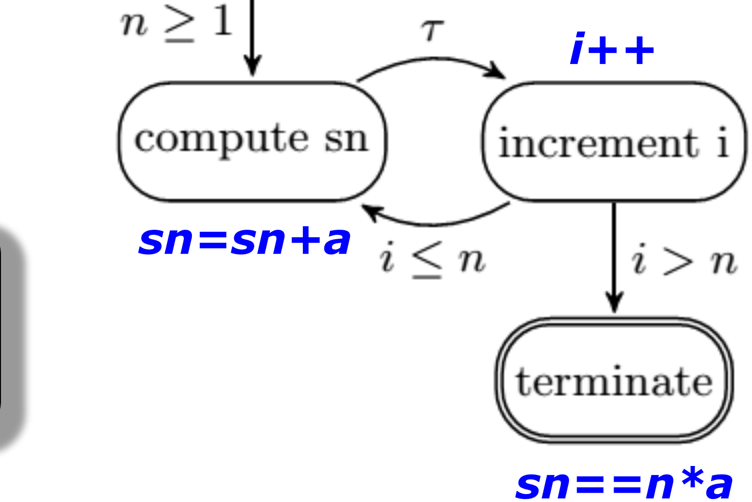
Difficulties in proving the correctness of programs with loops in BMC

- BMC techniques can falsify properties up to a given depth k
– they can prove correctness only if an upper bound of k is known (**unwinding assertion**)
  - » BMC tools typically fail to verify programs that contain bounded and unbounded loops

$$S_n = \sum_{i=1}^{n} a = na, n \geq 1$$

the loop will be unfolded 2^{n-1} times (in the worst case, 2^{32-1} times on 32 bits integer)

4,294,967,295 loop unwindings



BMC of Multi-threaded Software

- concurrency bugs are tricky to **reproduce/debug** because they usually occur under specific thread interleavings
  - most common errors: 67% related to atomicity and order violations, 30% related to deadlock [Lu et al.' 08]
- problem: the number of interleavings grows exponentially with the number of threads (n) and program statements (s)
  - number of executions: $O(n^s)$
  - context switches among threads increase the number of possible executions

BMC of single- and multi-threaded software

Bounded Model Checking of Software:

- symbolically executes programs into SSA, produces QF formulae
- unrolls loops and recursions up to a maximum bound *k*
- check whether corresponding formula is satisfiable
  - safety properties (array bounds, pointer dereferences, overflows,...)
  - user-specified properties

multi-threaded programs:

- combines explicit-state with symbolic model checking
- symbolic state hashing & monotonic POR
- context-bounded analysis (optional context bound)

Satisfiability Modulo Theories (1)

SMT decides the **satisfiability** of first-order logic formulae using the combination of different **background theories** (building-in operators)

Theory	**Example**
Equality	$x_1=x_2 \land \neg (x_1=x_3) \Rightarrow \neg(x_1=x_3)$
Bit-vectors	(b >> i) & 1 = 1
Linear arithmetic	$(4y_1 + 3y_2 \geq 4) \lor (y_2 - 3y_3 \leq 3)$
Arrays	$(j = k \land a[k]=2) \Rightarrow a[j]=2$
Combined theories	$(j \leq k \land a[j]=2) \Rightarrow a[i] < 3$

Satisfiability Modulo Theories (2)

- Given
  - a decidable $\sum$-theory T
  - a quantifier-free formula φ

  φ **is T-satisfiable** iff T $\cup$ $\{\varphi\}$ is satisfiable, i.e., there exists a structure that satisfies both formula and sentences of T
- Given
  - a set $\Gamma \cup \{\varphi\}$ of first-order formulae over T

  φ **is a T-consequence of** Γ ($\Gamma \models_T \varphi$) iff every model of T $\cup$ Γ is also a model of φ
- Checking $\Gamma \models_T \varphi$ can be reduced in the usual way to checking the T-satisfiability of $\Gamma \cup \{\neg\varphi\}$

Satisfiability Modulo Theories (3)

- let **a** be an array, **b**, **c** and **d** be signed bit-vectors of width 16, 32 and 32 respectively, and let **g** be an unary function.

$$g(select(store(a,c,12)), SignExt(b,16)+3)$$
$$\neq g(SignExt(b,16)-c+4) \wedge SignExt(b,16) = c-3 \wedge c+1 = d-4$$

⇩ **b'** extends **b** to the signed equivalent bit-vector of size 32

$$step\,1: g(select(store(a,c,12), b'+3)) \neq g(b'-c+4) \wedge b' = c-3 \wedge c+1 = d-4$$

⇩ replace b' by c−3 in the inequality

$$step\,2: g(select(store(a,c,12), c-3+3)) \neq g(c-3-c+4) \wedge c-3 = c-3 \wedge c+1 = d-4$$

⇩ using facts about bit-vector arithmetic

$$step\,3: g(select(store(a,c,12), c)) \neq g(1) \wedge c-3 = c-3 \wedge c+1 = d-4$$

Satisfiability Modulo Theories (4)

$$step\,3: g(select(store(a,c,12),c)) \neq g(1) \wedge c-3 = c-3 \wedge c+1 = d-4$$

applying the theory of arrays

$$step\,4: g(12) \neq g(1) \wedge c-3 \wedge c+1 = d-4$$

The function g implies that for all x and y,
if x = y, then g (x) = g (y) (*congruence rule*).

$$step\;5: SAT\,(\mathrm{c}=5,\,\mathrm{d}=10)$$

- SMT solvers also apply:
  - standard algebraic reduction rules $\boxed{r \wedge false \mapsto false}$
  - contextual simplification $\boxed{a = 7 \wedge p(a) \mapsto a = 7 \wedge p(7)}$