

Bug Catching: Automated Program Verification

15414/15614 Fall 2025

Lecture 1: Introduction

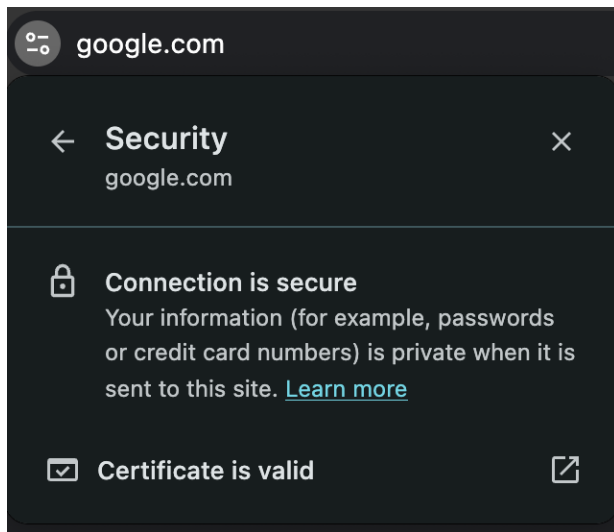
Matt Fredrikson

August 26, 2025

Learning objectives

For this lecture

- ▶ What is this course about?
- ▶ What are the learning objectives for the course?
- ▶ How does it fit into the curriculum?
- ▶ How does the course work?
- ▶ Getting set up for the course



► **Underlying tech: TLS**

- TLS (Transport Layer Security) is the successor to SSL (Secure Sockets Layer).
- It ensures confidentiality, integrity, and authentication for network communications.

► **Underlying tech: TLS**

- TLS (Transport Layer Security) is the successor to SSL (Secure Sockets Layer).
- It ensures confidentiality, integrity, and authentication for network communications.

► **Why does it matter?**

- Essential for establishing trust between users and online services.
- Most widely deployed security protocol on the Internet.

- ▶ **Underlying tech: TLS**

- ▶ TLS (Transport Layer Security) is the successor to SSL (Secure Sockets Layer).
- ▶ It ensures confidentiality, integrity, and authentication for network communications.

- ▶ **Why does it matter?**

- ▶ Essential for establishing trust between users and online services.
- ▶ Most widely deployed security protocol on the Internet.

- ▶ The most widely used *implementation* is OpenSSL.

- ▶ OpenSSL announced critical vulnerability in their implementation of the Heartbeat Extension.



- ▶ OpenSSL announced critical vulnerability in their implementation of the Heartbeat Extension.
- ▶ “The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software.”



- ▶ OpenSSL announced critical vulnerability in their implementation of the Heartbeat Extension.
- ▶ “The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software.”
- ▶ “...this allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.”



Heartbleed, explained



Image source: Randall Munroe, xkcd.com

Heartbleed, explained

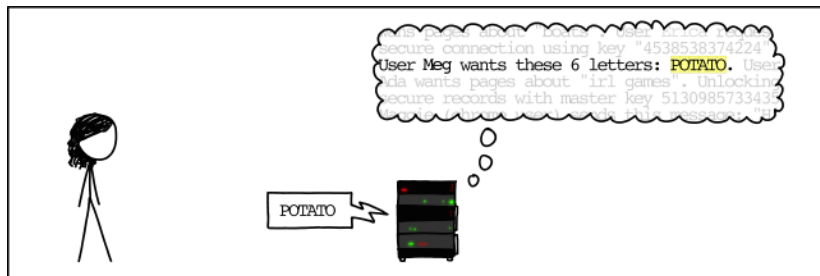


Image source: Randall Munroe, xkcd.com

Heartbleed, explained



Image source: Randall Munroe, xkcd.com

Heartbleed, explained

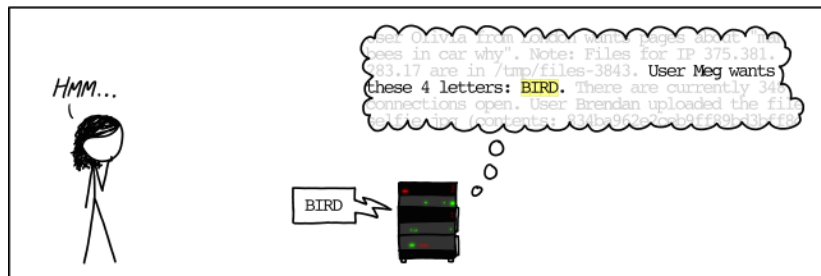


Image source: Randall Munroe, xkcd.com

Heartbleed, explained



Image source: Randall Munroe, xkcd.com

Heartbleed, explained

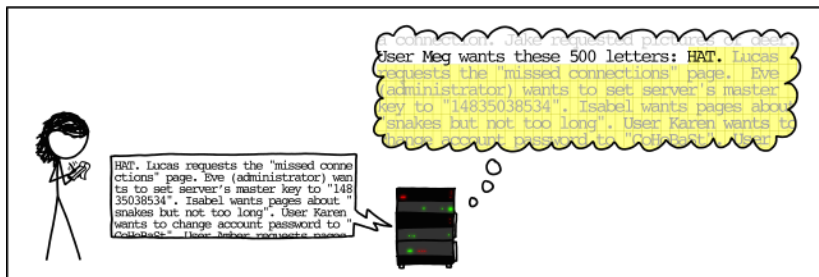


Image source: Randall Munroe, xkcd.com

The cost

- ▶ Estimates are ~\$500 million
 - ▶ Stolen data
 - ▶ Certificate revocation
 - ▶ Bandwidth
 - ▶ Engineering effort
 - ▶ ...

The cost

- ▶ Estimates are ~\$500 million
 - ▶ Stolen data
 - ▶ Certificate revocation
 - ▶ Bandwidth
 - ▶ Engineering effort
 - ▶ ...

Tech giants spend millions to stop another Heartbleed

© 25 April 2014

EXPLOITS AND VULNERABILITIES | NEWS

Five years later, Heartbleed vulnerability still unpatched

Posted: September 12, 2019 by Gilad Maayan

Algorithms vs. code

```
1 int binarySearch(int key, int[] a, int n) {
2     int low = 0;
3     int high = n;
4
5     while (low < high) {
6         int mid = (low + high) / 2;
7
8         if(a[mid] == key) return mid; // key found
9         else if(a[mid] < key) {
10             low = mid + 1;
11         } else {
12             high = mid;
13         }
14     }
15     return -1; // key not found.
16 }
```


This is a correct binary search algorithm

Code matters

This is a correct binary search algorithm

But if $\text{low} + \text{high} > 2^{31} - 1$

...then $\text{mid} = (\text{low} + \text{high}) / 2$ becomes negative

This is a correct binary search algorithm

But if $\text{low} + \text{high} > 2^{31} - 1$

...then $\text{mid} = (\text{low} + \text{high}) / 2$ becomes negative

- ▶ Best case: `ArrayIndexOutOfBoundsException`
- ▶ Worst case: undefined (that is, arbitrary) behavior

This is a correct binary search algorithm

But if $\text{low} + \text{high} > 2^{31} - 1$

...then $\text{mid} = (\text{low} + \text{high}) / 2$ becomes negative

- ▶ Best case: `ArrayIndexOutOfBoundsException`
- ▶ Worst case: undefined (that is, arbitrary) behavior

We execute code, not algorithms.

How do we fix it?

Solution: `mid = low + (high - low)/2`

The fix

```
1 int binarySearch(int key, int[] a, int n) {
2     int low = 0;
3     int high = n;
4
5     while (low < high) {
6         int mid = low + (high - low) / 2;
7
8         if(a[mid] == key) return mid; // key found
9         else if(a[mid] < key) {
10             low = mid + 1;
11         } else {
12             high = mid;
13         }
14     }
15     return -1; // key not found.
16 }
```

The fix

```
1 int binarySearch(int key, int[] a, int n)
2 //@requires 0 <= n && n <= \length(A);
3 {
4     int low = 0;
5     int high = n;
6
7     while (low < high) {
8         int mid = low + (high - low) / 2;
9
10        if(a[mid] == key) return mid; // key found
11        else if(a[mid] < key) {
12            low = mid + 1;
13        } else {
14            high = mid;
15        }
16    }
17    return -1; // key not found.
18 }
```

The fix

```
1  int binarySearch(int key, int[] a, int n)
2  //@requires 0 <= n && n <= \length(a);
3  /*@ensures (\result == -1 && !is_in(key, A, 0, n))
4  @    // (0 <= \result && \result < n
5  @    && A[\result] == key); @*/
6  {
7      int low = 0;
8      int high = n;
9
10     while (low < high) {
11         int mid = low + (high - low) / 2;
12
13         if(a[mid] == key) return mid; // key found
14         else if(a[mid] < key) {
15             low = mid + 1;
16         } else {
17             high = mid;
18         }
19     }
20     return -1; // key not found.
21 }
```


The fix

```
1  int binarySearch(int key, int[] a, int n)
2  //@requires 0 <= n && n <= \length(a);
3  //@requires is_sorted(a, 0, n);
4  /*@ensures (\result == -1 && !is_in(key, A, 0, n))
5  @      || (0 <= \result && \result < n
6          && A[\result] == key); @*/
7  {
8      int low = 0;
9      int high = n;
10
11     while (low < high) {
12         int mid = low + (high - low) / 2;
13
14         if(a[mid] == key) return mid; // key found
15         else if(a[mid] < key) {
16             low = mid + 1;
17         } else {
18             high = mid;
19         }
20     }
21     return -1; // key not found.
22 }
```

How do we know if it's correct?

How do we know if it's correct?

Test cases

- ▶ Incomplete
- ▶ Not always realistic
- ▶ ...

How do we know if it's correct?

Test cases

- ▶ Incomplete
- ▶ Not always realistic
- ▶ ...

Code review

- ▶ Inconsistent
- ▶ Labor intensive

How do we know if it's correct?

Test cases

- ▶ Incomplete
- ▶ Not always realistic
- ▶ ...

Code review

- ▶ Inconsistent
- ▶ Labor intensive

Prove it

Specification \iff Implementation

How do we know if it's correct?

Test cases

- ▶ Incomplete
- ▶ Not always realistic
- ▶ ...

Code review

- ▶ Inconsistent
- ▶ Labor intensive

Prove it

Specification \iff Implementation

- ▶ Write a **precise** specification
- ▶ Assign code **mathematical** meaning
- ▶ Use **logic** to establish correctness

Course objectives

- ▶ Formally specify correct behavior
- ▶ Develop mathematical semantics of programs
- ▶ Reason about program correctness
- ▶ Learn how to write correct software, from beginning to end
- ▶ Use automated tools that assist with verification
- ▶ Understand how verification tools work

Course objectives

- ▶ Formally specify correct behavior
- ▶ Develop mathematical semantics of programs
- ▶ Reason about program correctness
- ▶ Learn how to write correct software, from beginning to end
- ▶ Use automated tools that assist with verification
- ▶ Understand how verification tools work
- ▶ Make you better programmers

Part I: Reasoning about programs: from 122 and 150 to 414

- ▶ Gain intuitive understanding of language and methodology

Part I: Reasoning about programs: from 122 and 150 to 414

- ▶ Gain intuitive understanding of language and methodology

Part II: From informal to formal reasoning

- ▶ Specifying meaning of programs
- ▶ Specifying meaning of propositions
- ▶ Formal reasoning and its justification

Part I: Reasoning about programs: from 122 and 150 to 414

- ▶ Gain intuitive understanding of language and methodology

Part II: From informal to formal reasoning

- ▶ Specifying meaning of programs
- ▶ Specifying meaning of propositions
- ▶ Formal reasoning and its justification

Part III: Automated reasoning

- ▶ Automated proof techniques
- ▶ Model checking

Algorithmic approaches

Formal proofs are tedious

Automatic methods can:

- ▶ Check our work
- ▶ Fill in low-level details
- ▶ Give diagnostic info

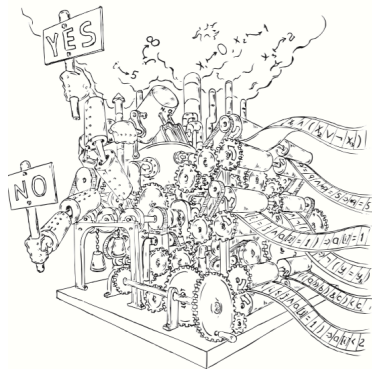


Image source: Daniel Kroening & Ofer
Strichman, *Decision Procedures*

Algorithmic approaches

Formal proofs are tedious

Automatic methods can:

- ▶ Check our work
- ▶ Fill in low-level details
- ▶ Give diagnostic info

They usually cannot:

- ▶ Verify *everything*
- ▶ Write specifications
- ▶ Fix bugs

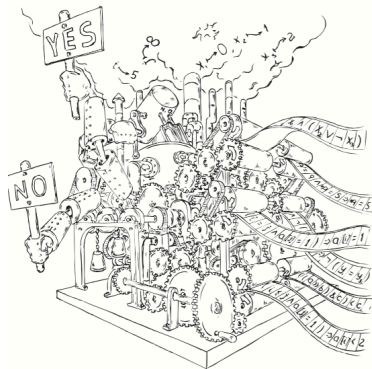


Image source: Daniel Kroening & Ofer Strichman, *Decision Procedures*

Algorithmic approaches

Formal proofs are tedious

Automatic methods can:

- ▶ Check our work
- ▶ Fill in low-level details
- ▶ Give diagnostic info

They usually cannot:

- ▶ Verify *everything*
- ▶ Write specifications
- ▶ Fix bugs

This is what you will learn!

- ▶ Make use of these methods
- ▶ How (and when) they work

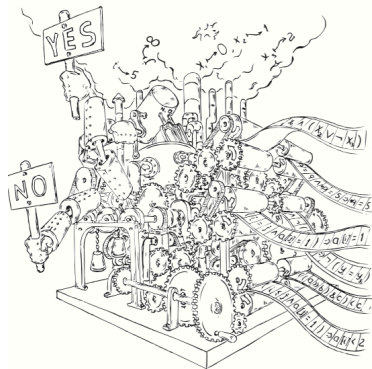


Image source: Daniel Kroening & Ofer Strichman, *Decision Procedures*

Different traditions and techniques

Functional programming: dependent types

- ▶ Proofs are expressed in programs (Agda)
- ▶ Proof tactics are expressed as programs (Coq)

Different traditions and techniques

Functional programming: dependent types

- ▶ Proofs are expressed in programs (Agda)
- ▶ Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- ▶ Properties are expressed in contracts
- ▶ Reduce correctness to logical propositions (verification condition)
- ▶ Use automated theorem provers to prove VC

Different traditions and techniques

Functional programming: dependent types

- ▶ Proofs are expressed in programs (Agda)
- ▶ Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- ▶ Properties are expressed in contracts
- ▶ Reduce correctness to logical propositions (verification condition)
- ▶ Use automated theorem provers to prove VC

Why3 (this course) supports both!

- ▶ Functional and imperative code in WhyML
- ▶ Automated provers for VC (Z3, CVC, alt-ergo, ...)
- ▶ Interactive provers for VC (Coq)

Reasoning about correctness

Functional Correctness

- ▶ Specification
- ▶ Proof

Reasoning about correctness

Functional Correctness

- ▶ Specification
- ▶ Proof

Specify behavior with logic

- ▶ Declarative
- ▶ Precise

Reasoning about correctness

Functional Correctness

- ▶ Specification
- ▶ Proof

Specify behavior with logic

- ▶ Declarative
- ▶ Precise

Systematic proof techniques

- ▶ Derived from semantics
- ▶ Exhaustive proof rules
- ▶ Often automatable

Reasoning about correctness

Functional Correctness

- ▶ Specification
- ▶ Proof

Specify behavior with logic

- ▶ Declarative
- ▶ Precise

Systematic proof techniques

- ▶ Derived from semantics
- ▶ Exhaustive proof rules
- ▶ Often automatable

```
1 int[] array_copy(int[] A, int n)
2 //@requires 0 <= n && n <= \length(A);
3 //@ensures \length(\result) == n;
4 {
5     int[] B = alloc_array(int, n);
6
7     for (int i = 0; i < n; i++)
8         //@loop_invariant 0 <= i;
9     {
10         B[i] = A[i];
11     }
12
13     return B;
14 }
```

Reasoning about correctness

Functional Correctness

- ▶ Specification
- ▶ Proof

Specify behavior with logic

- ▶ Declarative
- ▶ Precise

Systematic proof techniques

- ▶ Derived from semantics
- ▶ Exhaustive proof rules
- ▶ Often automatable

```
1 int[] array_copy(int[] A, int n)
2 //@requires 0 <= n && n <= \length(A);
3 //@ensures \length(\result) == n;
4 {
5     int[] B = alloc_array(int, n);
6
7     for (int i = 0; i < n; i++)
8         //@loop_invariant 0 <= i;
9         {
10             B[i] = A[i];
11         }
12
13     return B;
14 }
```

But ...

Deductive verification platform

- ▶ Programming language (WhyML, derived from OCaml)
- ▶ Verification toolchain

Deductive verification platform

- ▶ Programming language (WhyML, derived from OCaml)
- ▶ Verification toolchain

Rich specification language

- ▶ Pre- and post-conditions, loop invariants, assertions
- ▶ Pure mathematical functions
- ▶ Termination metrics

Deductive verification platform

- ▶ Programming language (WhyML, derived from OCaml)
- ▶ Verification toolchain

Rich specification language

- ▶ Pre- and post-conditions, loop invariants, assertions
- ▶ Pure mathematical functions
- ▶ Termination metrics

Programmer writes specification, partial annotations

Deductive verification platform

- ▶ Programming language (WhyML, derived from OCaml)
- ▶ Verification toolchain

Rich specification language

- ▶ Pre- and post-conditions, loop invariants, assertions
- ▶ Pure mathematical functions
- ▶ Termination metrics

Programmer writes specification, partial annotations

Compiler proves correctness

Deductive verification platform

- ▶ Programming language (WhyML, derived from OCaml)
- ▶ Verification toolchain

Rich specification language

- ▶ Pre- and post-conditions, loop invariants, assertions
- ▶ Pure mathematical functions
- ▶ Termination metrics

Programmer writes specification, partial annotations

Compiler proves correctness

When it works! (It's not quite like a type-checker ...)

Systems that prove that programs match their specifications

Basic idea:

1. Translate programs into *proof obligations*
2. Encode proof obligations as satisfiability
3. Solve using a *decision procedure*

Systems that prove that programs match their specifications

Basic idea:

1. Translate programs into *proof obligations*
2. Encode proof obligations as satisfiability
3. Solve using a *decision procedure*

Problem is undecidable!

1. Prover needs “hints” from programmer
2. Finding the right set of hints can be challenging

Systems that prove that programs match their specifications

Basic idea:

1. Translate programs into *proof obligations*
2. Encode proof obligations as satisfiability
3. Solve using a *decision procedure*

Problem is undecidable!

1. Prover needs “hints” from programmer
2. Finding the right set of hints can be challenging

Verifiers are complex systems

- ▶ We'll deep-dive into selected components
- ▶ Understand “big picture” for the rest

Recent developments



Recent developments



Implement a ``sat`` function in Why3 that matches this signature:

```
sat (cnf : cnf) : option valuation
```

If ``cnf`` is satisfiable, then ``sat`` should return ``Some rho``, where ``rho`` is a satisfying assignment for ``cnf``. Otherwise, it should return ``None``.

Recent developments

```
let rec sat (cnf : cnf) : option valuation =
  ensures {
    match result with
    | None -> unsat cnf
    | Some rho -> sat_with rho cnf
  }
let rho = Array.make cnf.nvars False in
let rec dfs (n : int) : bool =
  requires { valid_valuation rho cnf }
  requires { 0 <= n <= cnf.nvars }
  variant { cnf.nvars - n }
  ensures {
    if result == true then
      sat_with rho cnf
    else
      forall rho':valuation. array_eq_sub rho' rho 0 n -> not (sat_with rho' cnf)
  }
```

Recent developments

MF

The ``dfs`` function needs additional specifications to prove that the array accesses are in bounds.

Recent developments

MF

The `dfs` function needs additional specifications to prove that the array accesses are in bounds.



The following is an updated version of the `dfs` function with added specifications:

 Copy code

```
let rec dfs (rho : valuation) (n : int) : bool =  
  requires { valid_valuation rho cnf }  
  requires { 0 <= n < cnf.nvars }  
  variant { n }  
  ensures {  
    if result == true then  
      sat_with rho cnf  
    else  
      forall i:int. 0 <= i <= n -> not (sat_with (set rho i true) cnf)  
    }  
}
```

Recent developments

MF

The ``ensures`` specification on ``dfs`` is not correct. Specifying ``sat_with (set rho i true) cnf`` for all ``i`` in the range from ``0`` to ``n`` is not relevant; what is relevant is that if ``result <> true``, then ``unsat cnf``.

Recent developments

MF

The ``ensures`` specification on ``dfs`` is not correct. Specifying ``sat_with (set rho i true) cnf`` for all ``i`` in the range from ``0`` to ``n`` is not relevant; what is relevant is that if ``result <> true``, then ``unsat cnf``.



The following is an updated version of the ``dfs`` function with a corrected ``ensures`` specification:

 Copy code

```
let rec dfs (rho : valuation) (n : int) : bool =  
  requires { valid_valuation rho cnf }  
  requires { 0 <= n < cnf.nvars }  
  variant { n }  
  ensures {  
    if result == true then  
      sat_with rho cnf  
    else  
      unsat cnf  
  }
```

(Limitations of) recent developments

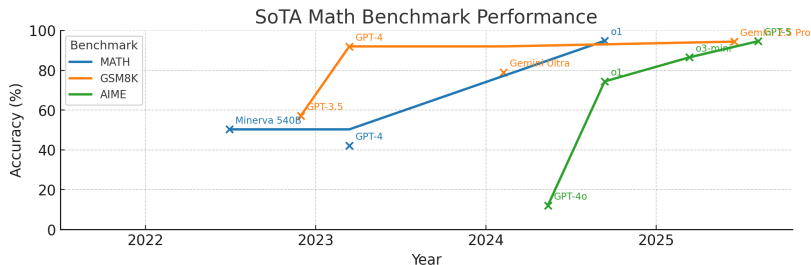
Eventually, ChatGPT produced a verified implementation

Getting there took expertise!

(Limitations of) recent developments

Eventually, ChatGPT produced a verified implementation

Getting there took expertise!



(Limitations of) recent developments

Eventually, ChatGPT produced a verified implementation

Getting there took expertise!



(Limitations of) recent developments

Eventually, ChatGPT produced a verified implementation

Getting there took expertise!

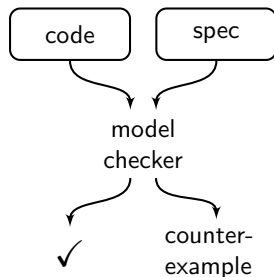
If you use a code model/assistant to complete a major task:

- ▶ Document your approach / save the transcript
- ▶ Describe the relevant parts in your solution
- ▶ Come talk to me about it

Model Checking

Fully-automatic techniques for finding bugs (or proving their absence)

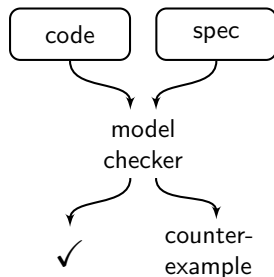
- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples



Model Checking

Fully-automatic techniques for finding bugs (or proving their absence)

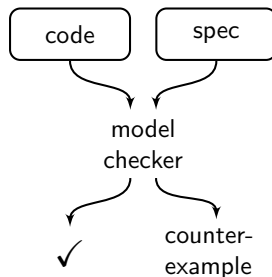
- ▶ Specifications written in *propositional temporal logic*
- ▶ Verification by exhaustive state space search
- ▶ Diagnostic counterexamples
- ▶ No proofs!



Model Checking

Fully-automatic techniques for finding bugs (or proving their absence)

- ▶ Specifications written in *propositional temporal logic*
- ▶ Verification by exhaustive state space search
- ▶ Diagnostic counterexamples
- ▶ No proofs!
- ▶ **Downside:** “State explosion”



Model Checking

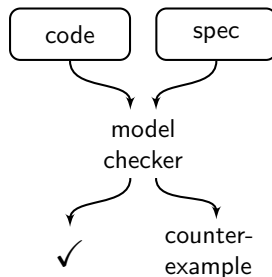
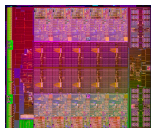
Fully-automatic techniques for finding bugs (or proving their absence)

- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples
- No proofs!
- **Downside:** “State explosion”

10^{70} atoms



10^{500000} states



Clever ways of dealing with state explosion:

Clever ways of dealing with state explosion:

- ▶ Partial order reduction
- ▶ Bounded model checking
- ▶ Symbolic representations
- ▶ Abstraction & refinement

Model Checking

Clever ways of dealing with state explosion:

- ▶ Partial order reduction
- ▶ Bounded model checking
- ▶ Symbolic representations
- ▶ Abstraction & refinement

Now widely used for bug-finding:

- ▶ Hardware, software, protocols, ...
- ▶ Microsoft, Intel, Amazon, Google, NASA,
...

Model Checking

Clever ways of dealing with state explosion:

- ▶ Partial order reduction
- ▶ Bounded model checking
- ▶ Symbolic representations
- ▶ Abstraction & refinement

Now widely used for bug-finding:

- ▶ Hardware, software, protocols, ...
- ▶ Microsoft, Intel, Amazon, Google, NASA,
...



Ed Clarke, 1945–2020
Turing Award, 2007

Model Checking

Clever ways of dealing with state explosion:

- ▶ Partial order reduction
- ▶ Bounded model checking
- ▶ Symbolic representations
- ▶ Abstraction & refinement

Now widely used for bug-finding:

- ▶ Hardware, software, protocols, ...
- ▶ Microsoft, Intel, Amazon, Google, NASA,
...



Ed Clarke, 1945–2020
Turing Award, 2007
First developed this
course!

6 assignments
done individually

Breakdown:

- ▶ 50% assignments
(written + programming)
- ▶ 15% mini-project 1
- ▶ 15% mini-project 2
- ▶ 20% final exam

2 mini-projects
pick from small menu
can work with a partner

Participation:

- ▶ Come to lecture
- ▶ Answer questions
(in class and on Piazza!)
- ▶ Contribute to discussion

Written parts of assignments

Written homeworks focus on theory and fundamental skills

Written parts of assignments

Written homeworks focus on theory and fundamental skills

Grades are based on:

- ▶ Correctness of your answer
- ▶ How you present your reasoning

Written parts of assignments

Written homeworks focus on theory and fundamental skills

Grades are based on:

- ▶ Correctness of your answer
- ▶ How you present your reasoning

Strive for **clarity & conciseness**

- ▶ Show each step of your reasoning
- ▶ State your assumptions
- ▶ Answers without these → no points

Programming parts of assignments

For the programming, you will:

- ▶ Implement some functionality (data structure or algorithm)
- ▶ Specify correctness for that functionality
- ▶ Use Why3 to prove it correct

Programming parts of assignments

For the programming, you will:

- ▶ Implement some functionality (data structure or algorithm)
- ▶ Specify correctness for that functionality
- ▶ Use Why3 to prove it correct

Most important criterion is **correctness**.

Programming parts of assignments

For the programming, you will:

- ▶ Implement some functionality (data structure or algorithm)
- ▶ Specify correctness for that functionality
- ▶ Use Why3 to prove it correct

Most important criterion is **correctness**.

Full points when you provide the following

- ▶ Correct implementation
- ▶ Correct specification
- ▶ Correct contracts
- ▶ Sufficient contracts for verification

Programming parts of assignments

For the programming, you will:

- ▶ Implement some functionality (data structure or algorithm)
- ▶ Specify correctness for that functionality
- ▶ Use Why3 to prove it correct

Most important criterion is **correctness**.

Full points when you provide the following

- ▶ Correct implementation
- ▶ Correct specification
- ▶ Correct contracts
- ▶ Sufficient contracts for verification

Partial credit depending on how many of these you achieve

Clarity & conciseness is necessary for partial credit!

Mini-Projects

Mini-projects are intended to build proficiency in:

- ▶ Writing good specifications
- ▶ Applying course principles to practice
- ▶ Making effective use of automated tools
- ▶ **Writing useful & correct code**

Mini-Projects

Mini-projects are intended to build proficiency in:

- ▶ Writing good specifications
- ▶ Applying course principles to practice
- ▶ Making effective use of automated tools
- ▶ **Writing useful & correct code**

Gradual progression to sophistication:

1. Familiarize yourself with Why3
2. Implement and prove something
3. Work with more complex data structures
4. Implement and prove something really interesting
5. Optimize your implementation, still verified

Late Policy

Late days

- ▶ 5 late days to use throughout the semester
- ▶ No more than 2 late days on any assignment
- ▶ Late days do not apply to mini-projects!

Website: <http://www.cs.cmu.edu/~15414>

Course staff contact: Piazza

Lecture: Tuesdays & Thursdays, 2-3:20pm

Office Hours: TBD, schedule on website and course calendar soon

Assignments: Gradescope