Bug Catching: Automated Program Verification
15414/15614 Fall 2017
Lecture 1:
Introduction

Matt Fredrikson, André Platzer
{mfredrik, aplatzer}@cs

August 29, 2017
Course Staff

Matt Fredrikson
Instructor

André Platzer
Instructor

Jonathan Laurent
TA

Tianyu Li
TA
Does the software do what it is supposed to do?
What happens when software misbehaves

April, 2014

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."

Matt Fredrikson

Model Checking
What happens when software misbehaves

- **April, 2014** OpenSSL announced critical vulnerability in their implementation of the Heartbeat Extension.
What happens when software misbehaves

- **April, 2014** 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.”
April, 2014 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

User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoRaSt". User Adam requests pages.

Image source: Randall Munroe, xkcd.com
Does this do what it is supposed to?

```c
int binarySearch(int key, int[] a, int n) {
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = (low + high) / 2;

        if(a[mid] == key) return mid; // key found
        else if(a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return -1; // key not found.
}
```
int binarySearch(int key, int[] a, int n) {
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = (low + high) / 2;

        if (a[mid] == key) return mid; // key found
        else if (a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }

    return -1; // key not found.
This is a correct binary search algorithm.
This is a correct binary search algorithm.

But what if \( \text{low} + \text{high} > 2^{31} - 1 \)?
This is a correct binary search algorithm.

But what 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 what if \( \text{low} + \text{high} > 2^{31} - 1 \)?

Then \( \text{mid} = (\text{low} + \text{high}) / 2 \) becomes negative

- Best case: ArrayIndexOutOfBoundsException
- Worst case: undefined behavior
This is a correct binary search algorithm.

But what if \( \text{low} + \text{high} \geq 2^{31} - 1? \)

Then \( \text{mid} = (\text{low} + \text{high}) / 2 \) becomes negative

- **Best case:** ArrayIndexOutOfBoundsException
- **Worst case:** undefined behavior
This is a correct binary search algorithm.

But what if \( \text{low} + \text{high} > 2^{31} - 1 \)?

Then \( \text{mid} = (\text{low} + \text{high}) / 2 \) becomes negative
  - **Best case:** `ArrayIndexOutOfBoundsException`
  - **Worst case:** undefined behavior

Algorithm may be correct. The code, another story...
How do we fix it?

The culprit: \( \text{mid} = \frac{\text{low} + \text{high}}{2} \)
How do we fix it?

The culprit: mid = (low + high) / 2

Need to make sure we don’t overflow at any point
How do we fix it?

The culprit: \( \text{mid} = (\text{low} + \text{high}) / 2 \)

Need to make sure we don’t overflow at any point

Solution: \( \text{mid} = \text{low} + (\text{high} - \text{low})/2 \)
```c
int binarySearch(int key, int[] a, int n) {
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = low + (high - low) / 2;

        if(a[mid] == key) return mid;  // key found
        else if(a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }

    return -1;  // key not found.
}
```
```c
int binarySearch(int key, int[] a, int n) {
    //@requires 0 <= n && n <= \length(A);
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = low + (high - low) / 2;
        if(a[mid] == key) return mid; // key found
        else if(a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return -1; // key not found.
}
```
int binarySearch(int key, int[] a, int n)
//@requires 0 <= n && n <= \length(a);
//@requires is_sorted(a, 0, n);
/*@ensures (@result == -1 && !is_in(key, A, 0, n))
    @ (0 <= @result, @result < n)
    @ A[@result] == key); @*/
{
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = low + (high - low) / 2;

        if(a[mid] == key) return mid; // key found
        else if(a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return -1; // key not found.
How do we know if it’s correct?

One solution: test the code

Possibly incomplete!

Exhaustive testing usually not feasible

Better: prove that it’s correct

Specifications must be precise, unambiguous

Meaning of code must be well-defined

When done well, gives strong indication of correctness

Specifications must be validated

Proofs must be correct

Reasoning must be sound
How do we know if it’s correct?

One solution: test the code
How do we know if it’s correct?

One solution: test the code
- Possibly incomplete → uncertain answer
- Exhaustive testing usually not feasible
How do we know if it’s correct?

One solution: test the code

- Possibly incomplete $\rightarrow$ uncertain answer
- Exhaustive testing usually not feasible

Better: **prove** that that it’s correct

Specification $\iff$ Implementation
How do we know if it’s correct?

One solution: test the code

- Possibly incomplete \(\rightarrow\) uncertain answer
- Exhaustive testing usually not feasible

Better: **prove** that that it’s correct

\[\text{Specification} \iff \text{Implementation}\]

- Specifications must be precise, unambiguous
- Meaning of code must be well-defined
How do we know if it’s correct?

One solution: test the code

- Possibly incomplete $\rightarrow$ uncertain answer
- Exhaustive testing usually not feasible

Better: **prove** that it’s correct

\[ \text{Specification} \iff \text{Implementation} \]

- Specifications must be precise, unambiguous
- Meaning of code must be well-defined

When done well, gives strong indication of correctness
How do we know if it’s correct?

One solution: test the code
  ▶ Possibly incomplete → uncertain answer
  ▶ Exhaustive testing usually not feasible

Better: **prove** that that it’s correct

\[ \text{Specification} \iff \text{Implementation} \]

▶ Specifications must be precise, unambiguous
▶ Meaning of code must be well-defined

When done well, gives strong indication of correctness
  ▶ Specifications must be validated
  ▶ Proofs must be correct
  ▶ Reasoning must be sound
Algorithmic Approaches

Formal proofs are tedious, labor-intensive
Algorithmic Approaches

Formal proofs are tedious, labor-intensive

We want algorithms to:
- Check our work
- Fill in low-level details
- Give diagnostic info
- Verify the system (if possible)
Formal proofs are tedious, labor-intensive

We want algorithms to:

- Check our work
- Fill in low-level details
- Give diagnostic info
- Verify the system (if possible)

This is called \textit{algorithmic verification}
Formal proofs are tedious, labor-intensive

We want algorithms to:
- Check our work
- Fill in low-level details
- Give diagnostic info
- Verify the system (if possible)

This is called *algorithmic verification*
Course objectives

▶ Identify and formalize program correctness
▶ Understand the formal semantics of programs
▶ Apply mathematical reasoning to program correctness
▶ Learn how to write correct software, from beginning to end
▶ Use automated tools that assist verifying your code
▶ Understand how verification tools work
Course objectives

- Identify and formalize program correctness
Course objectives

- Identify and formalize program correctness
- Understand the formal semantics of programs
Course objectives

- Identify and formalize program correctness
- Understand the formal semantics of programs
- Apply mathematical reasoning to program correctness
Course objectives

- Identify and formalize program correctness
- Understand the formal semantics of programs
- Apply mathematical reasoning to program correctness
- Learn how to write correct software, from beginning to end
Course objectives

- Identify and formalize program correctness
- Understand the formal semantics of programs
- Apply mathematical reasoning to program correctness
- Learn how to write correct software, from beginning to end
- Use automated tools that assist verifying your code
Course objectives

- Identify and formalize program correctness
- Understand the formal semantics of programs
- Apply mathematical reasoning to program correctness
- Learn how to write correct software, from beginning to end
- Use automated tools that assist verifying your code
- Understand how verification tools work
Reasoning about correctness

```
int[] array_copy(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures \length(\result) == n;
{
    int[] B = alloc_array(int, n);
    for (int i = 0; i < n; i++)
        //@loop_invariant 0 <= i;
        { B[i] = A[i]; }

    return B;
}
```
Reasoning about correctness

Functional Correctness

- Specification
- Proof

```c
int[] array_copy(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures \length(result) == n;
{
    int[] B = alloc_array(int, n);
    for (int i = 0; i < n; i++)
        //@loop_invariant 0 <= i;
        
        B[i] = A[i];
    
    return B;
}
```
Reasoning about correctness

Functional Correctness

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise
- Amenable to proof

```java
int[] array_copy(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures \length(result) == n;
{
    int[] B = alloc_array(int, n);
    for (int i = 0; i < n; i++)
        //@loop_invariant 0 <= i;
        { B[i] = A[i]; }
    return B;
}
```
Reasoning about correctness

**Functional Correctness**

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise
- Amenable to proof

Systematic proof techniques

- Based on language semantics
- Exhaustive proof rules
- Ideally, automatable

```c
int[] array_copy(int[] A, int n);
//@requires 0 <= n && n <= \length(A);
//@ensures \length(\result) == n;
{
    int[] B = alloc_array(int, n);
    for (int i = 0; i < n; i++)
        //@loop_invariant 0 <= i;
        { B[i] = A[i]; }
    return B;
}
```
Deductive verification platform

- Programming language
- Automated verification tools
Deductive verification platform
  ▶ Programming language
  ▶ Automated verification tools

Rich specification language
  ▶ Pre and postconditions, assertions
  ▶ Pure mathematical functions
  ▶ Termination metrics
Deductive verification platform
  ▶ Programming language
  ▶ Automated verification tools

Rich specification language
  ▶ Pre and postconditions, assertions
  ▶ Pure mathematical functions
  ▶ Termination metrics

Programmer writes specification, proof annotations

Compiler checks correctness automatically*!
let binary_search (a : array int) (v : int) {
  requires { sorted(a) }
  ensures { 0 <= result < length a && a[result] = v }
  raises { Not_found -> forall i:int. 0 <= i < length a -> a[i] <> v }
  = try
    let l = ref 0 in
    let u = ref (length a - 1) in
    while !l <= !u do
      invariant { 0 <= !l \&\& !u < length a }
      invariant { forall i : int. 0 <= i < length a -> a[i] = v -> !l <= i <= !u }
      variant { !u - !l }
      let m = !l + div (!u - !l) 2 in
      assert { !l <= m <= !u };
      if a[m] < v then
        l := m + 1
      else if a[m] > v then
        u := m - 1
      else
        raise (Break m)
    done;
    raise Not_found
  with Break i ->
    i
  end}
Automated Verification

Algorithms for proving that programs match their specifications
Automated Verification

Algorithms for proving 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
Automated Verification

Algorithms for proving that programs match their specifications

Problem is undecidable!
1. Require annotations
2. Relieve manual burden by inferring some annotations

Basic idea:
1. Translate programs into *proof obligations*
2. Encode proof obligations as satisfiability
3. Solve using a decision procedure
Automated Verification

Algorithms for proving that programs match their specifications

Problem is undecidable!
1. Require annotations
2. Relieve manual burden by inferring some annotations

Verifiers are non-trivial tools

Basic idea:
1. Translate programs into proof obligations
2. Encode proof obligations as satisfiability
3. Solve using a decision procedure
Automatic techniques for finding bugs (or proving their absence)
Automatic techniques for finding bugs (or proving their absence)
Automatic techniques for finding bugs (or proving their absence)

- Specifications written in propositional temporal logic

<table>
<thead>
<tr>
<th>code</th>
<th>spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>model checker</td>
<td></td>
</tr>
</tbody>
</table>

☑ counter-example
Automatic techniques for finding bugs (or proving their absence)

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

- Downside: “State explosion”

10^70 atoms
10^500000 states
**Automatic techniques for finding bugs (or proving their absence)**

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

---

![Diagram showing the process of model checking](image)

**Model Checking**

[Diagram showing the process of model checking]

- **code**
- **spec**

**model checker**

✓ **counter-example**
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!
**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  \quad 10^{500000} states
Clever ways of dealing with state explosion:
Clever ways of dealing with state explosion:

- Partial order reduction
- Bounded model checking
- Symbolic exploration
- Abstraction & refinement
Clever ways of dealing with state explosion:

- Partial order reduction
- Bounded model checking
- Symbolic exploration
- Abstraction & refinement

Now widely used for verification & bug-finding:

- Hardware, software, protocols, …
- Microsoft, Intel, Cadence, IBM, NASA, …
Clever ways of dealing with state explosion:

- Partial order reduction
- Bounded model checking
- Symbolic exploration
- Abstraction & refinement

Now widely used for verification & bug-finding:

- Hardware, software, protocols, …
- Microsoft, Intel, Cadence, IBM, NASA, …

Ed Clarke
Turing Award, 2007
Breakdown:
- 40% labs
- 25% written homework
- 30% exams (15% each, midterm and final)
- 5% participation

5 labs
Weekly written homework
In-class exams, closed-book

Participation:
- Come to lecture
- Ask questions, give answers
- Contribute to discussion
Grading

For the labs, you will:

- Implement some functionality (usually)
- Specify correctness for that functionality
- Prove it correct by annotating your implementation

Most important criterion is correctness.

Full points when you provide the following:

- Correct implementation
- Correct specification
- Correct annotations
- Sufficient annotations for verification

Partial credit depending on how many of these you achieve.

Clarity & conciseness is necessary for partial credit!
Grading

For the labs, you will:

► Implement some functionality (usually)
► Specify correctness for that functionality
► Prove it correct by annotating your implementation

Most important criterion is **correctness**.
For the labs, you will:

▶ Implement some functionality (usually)
▶ Specify correctness for that functionality
▶ Prove it correct by annotating your implementation

Most important criterion is correctness.

Full points when you provide the following

▶ Correct implementation
▶ Correct specification
▶ Correct annotations
▶ Sufficient annotations for verification
For the labs, you will:

▶ Implement some functionality (usually)
▶ Specify correctness for that functionality
▶ Prove it correct by annotating your implementation

Most important criterion is correctness.

Full points when you provide the following

▶ Correct implementation
▶ Correct specification
▶ Correct annotations
▶ Sufficient annotations for verification

Partial credit depending on how many of these you achieve

Clarity & conciseness is necessary for partial credit!
Grading

Written homeworks focus on theory and fundamental skills
Grading

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 well-explained reasoning don't count!
Grading

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 well-explained reasoning don’t count!
Late Policy

No late days on written homework

- Not intended to be time-intensive
- 25% deduction for each day past deadline

Can earn back missed points for proofs on labs

- Must submit original lab by the deadline
- Resubmit once within three days of deadline
- If proof is complete & correct, earn back points only on the proof
Late Policy

No late days on written homework
- Not intended to be time-intensive
- 25% deduction for each day past deadline

Can earn back missed points for proofs on labs
- Must submit original lab by the deadline
- Resubmit once within three days of deadline
- If proof is complete & correct, earn back points only on the proof
Website: http://www.cs.cmu.edu/~15414

Course staff contact: Piazza or 15414-staff@lists.andrew.cmu.edu

Lecture: Tuesdays & Thursdays, 10:30-11:50 GHC 4211

Matt Fredrikson, André Platzer
  ▶ Location: CIC 2126, GHC 9103
  ▶ Office Hours: TBD
  ▶ Email: mfredrik@cs, aplatzer@cs

Jonathan Laurent, Tianyu Li
  ▶ Office Hours: TBD
  ▶ Email: jonathan.laurent@cs, tli2@cs