Bug Catching: Automated Program Verification
15414/15614 Spring 2020
Lecture 1: Introduction

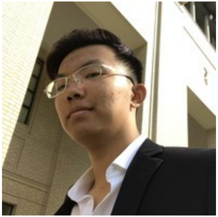
Matt Fredrikson

January 14, 2020

Course Staff

Matt Fredrikson
Instructor

Di Wang
TA



Ryan Chen
TA

Bad code

- **April, 2014** OpenSSL announced critical vulnerability in their implementation of the Heartbeat Extension.

Bad code

- **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.”

Bad code

- **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

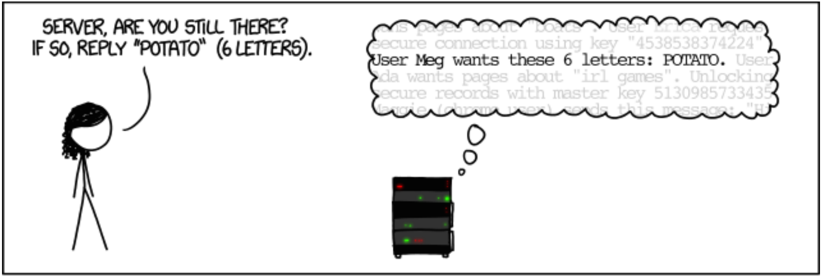


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

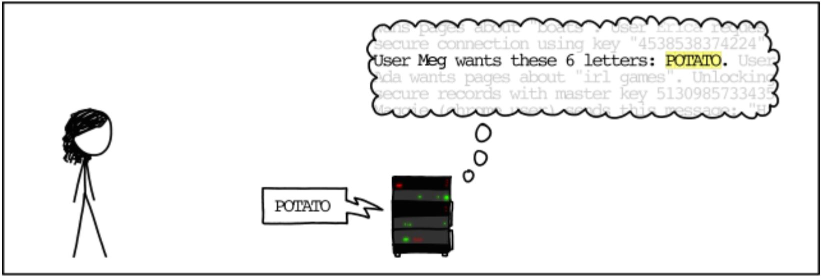


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

Image source: Randall Munroe, xkcd.com

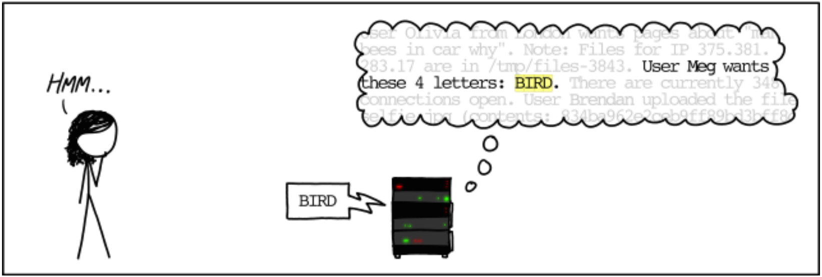


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

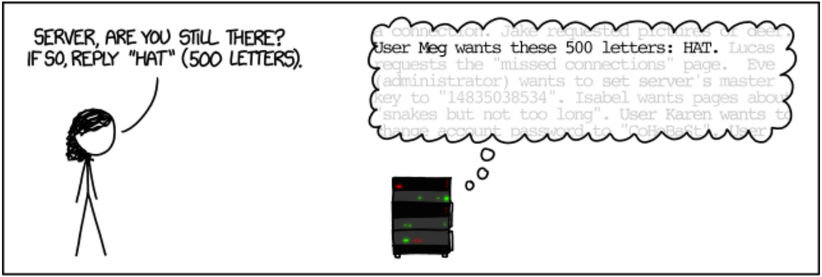


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

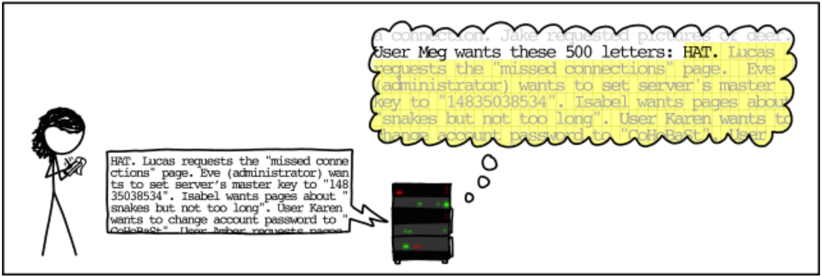


Image source: Randall Munroe, xkcd.com

Algorithms vs. code

```
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.
}
```

Code Matters

This is a correct binary search algorithm.

Code Matters

This is a correct binary search algorithm.

But what if `low + high` $> 2^{31} - 1$?

Code Matters

This is a correct binary search algorithm.

But what if $\texttt{low + high} > 2^{31} - 1$?

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

Code Matters

This is a correct binary search algorithm.

But what if `low + high` $> 2^{31} - 1$?

Then `mid = (low + high) / 2` becomes negative

- Best case: `ArrayIndexOutOfBoundsException`

Code Matters

This is a correct binary search algorithm.

But what if `low + high` $> 2^{31} - 1$?

Then `mid = (low + high) / 2` becomes negative

- Best case: `ArrayIndexOutOfBoundsException`
- Worst case: undefined behavior

Code Matters

This is a correct binary search algorithm.

But what if `low + high` $> 2^{31} - 1$?

Then `mid = (low + high) / 2` becomes negative

- Best case: `ArrayIndexOutOfBoundsException`
- Worst case: undefined behavior

Algorithm may be correct. But we run code, not algorithms.

How do we fix it?

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

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

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

The fix

```
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.
}
```

The fix

```
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.
}
```

The fix

```
int binarySearch(int key, int[] a, int n)
//@requires 0 <= n && n <= \length(a);
/*@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.
}
```

The fix

```
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?

How do we know if it’s correct?

One solution: testing

How do we know if it's correct?

One solution: testing

- Probably incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing not feasible

How do we know if it’s correct?

One solution: testing

- Probably incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing not feasible

Another: code review

How do we know if it's correct?

One solution: testing

- Probably incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing not feasible

Another: code review

- Correctness definitely important, but not the only thing
- Humans are fallable, bugs are subtle
- What's the specification?

How do we know if it's correct?

One solution: testing

- Probably incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing not feasible

Another: code review

- Correctness definitely important, but not the only thing
- Humans are fallable, bugs are subtle
- What's the specification?

Better: prove correctness

$$\mathit{Specification} \Longleftrightarrow \mathit{Implementation}$$

How do we know if it's correct?

One solution: testing

- Probably incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing not feasible

Another: code review

- Correctness definitely important, but not the only thing
- Humans are fallable, bugs are subtle
- What's the specification?

Better: prove correctness

$$\textit{Specification} \Longleftrightarrow \textit{Implementation}$$

- Specification must be precise
- Meaning of code must be well-defined
- Reasoning must be sound

Algorithmic Approaches

Formal proofs are tedious

Automatic methods can:

- Check our work
- Fill in low-level details
- Give diagnostic info
- Verify everything for us

This is what you will learn!

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

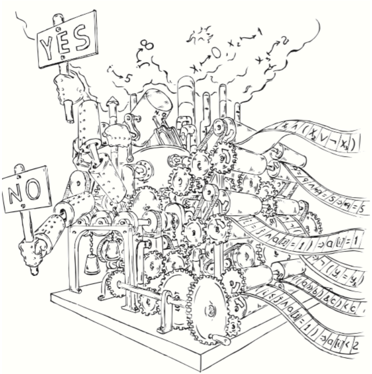


Image source: Daniel Kroening & Ofer Strichman, *Decision Procedures*

Course objectives

- Identify and formalize program correctness
- Understand language semantics
- 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

Functional Correctness

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise

Systematic proof techniques

- Derived from semantics
- Exhaustive proof rules
- Automatable*

```
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;
}
```

Why3

Deductive verification platform

- Programming language
- Verification toolchain

Rich specification language

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

Programmer writes specification, partial annotations

Compiler proves correctness automatically!

Automated Verifiers

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

Automated Verifiers

Systems that prove 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 Verifiers

Systems that prove that programs match their specifications

Problem is undecidable!

1. Require annotations
2. Relieve manual burden by inferring some annotations

Verifiers are complex systems

- We'll deep-dive into selected components
- Understand "big picture" for the rest

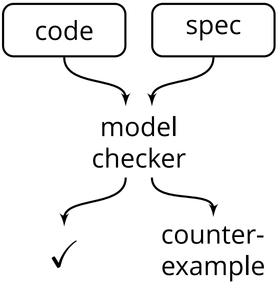
Basic idea:

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

Model Checking

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

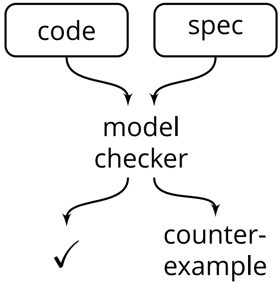
- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples



Model Checking

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

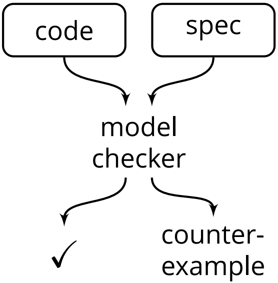
- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples
- No proofs!



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"

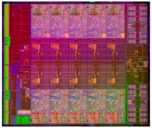


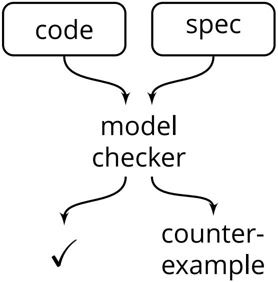
Model Checking

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:

Model Checking

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 verification & 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 verification & bug-finding:

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

Ed Clarke
Turing Award, 2007

Grading

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

Grading

For the labs, you will:

- Implement some functionality
- Specify correctness for that functionality
- Use Why3 to prove it correct

Grading

For the labs, you will:

- Implement some functionality
- Specify correctness for that functionality
- Use Why3 to prove it correct

Most important criterion is **correctness**.

Grading

For the labs, you will:

- Implement some functionality
- 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 annotations
- Sufficient annotations for verification

Grading

For the labs, you will:

- Implement some functionality
- 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 annotations
- Sufficient annotations for verification

Partial credit depending on how many of these you achieve

Clarity & conciseness is necessary for partial credit!

Labs

Labs are intended to build proficiency in:

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

Labs

Labs 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

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

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 these $\longrightarrow$ no points

Late Policy

No late days on written homework

- Not intended to be time-intensive
- 20% deduction for each day past deadline

Late Policy

No late days on written homework

- Not intended to be time-intensive
- 20% 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 (read: compiler accepts it), earn back points *only on the proof*

Late Policy

No late days on written homework

- Not intended to be time-intensive
- 20% 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 (read: compiler accepts it), earn back points *only on the proof*

Regrades

- No push-button regrades
- Discuss suspected error with us in person
- We will regrade the entire assignment

Logistics

Website: `http://www.cs.cmu.edu/~15414`

Course staff contact: Piazza or
`15414-staff@lists.andrew.cmu.edu`
Lecture: Tuesdays & Thursdays, 12:00-1:20pm GHC 4307

Matt Fredrikson

- Location: CIC 2126
- Office Hours: Thursdays 4pm

Di Wang, Ryan Chen

- Office Hours: TBD