CPSC 593L: Topics in Programming Languages

Automated Testing, Bug Detection, and Program Analysis

September 7th, 2022

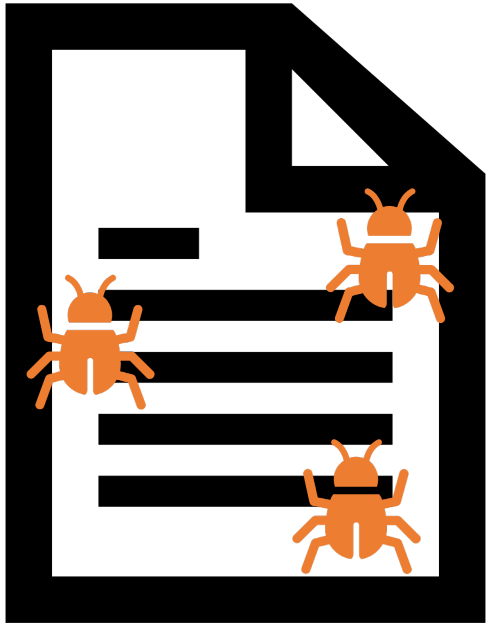
Instructor: Caroline Lemieux

Term: 2022W1

Class website: carolemieux.com/teaching/CPSC539L_2022w1.html

Sign up for class piazza: piazza.com/ubc.ca/winterterm12022/cpsc539l

Software Has Bugs



Bugs Have Increasing Consequences

Bugs Have Increasing Consequences

Badlock
Cloudbleed
Dirty COW
GHOST
Heartbleed
StageFright
ShellShock

Bugs in OpenSSL

Heartbleed

Severity: 7.5 HIGH
Introduced: 14 Mar 2012
Discovered: 1 Apr 2014
Fixed: 7 Apr 2014

```
"... can be used to reveal up
to 64k of memory to a connected
client or server ..."
```

Costs:

- >$500 million
- 30,000 X.509 certificates compromised
- 4.5 million patient records compromised
- CRA website shutdown, 900 SINs leaked
- ...

Bugs in OpenSSL

Heartbleed

Severity: 7.5 HIGH
Introduced: 14 Mar 2012
Discovered: 1 Apr 2014
Fixed: 7 Apr 2014

```
"... can be used to reveal up
to 64k of memory to a connected
client or server ..."
```

Costs:
- >$500 million
- 30,000 X.509 certificates compromised
- 4.5 million patient records compromised
- CRA website shutdown, 900 SINs leaked
- …

Bugs in OpenSSL

Heartbleed

Severity: 7.5 HIGH
Introduced: 14 Mar 2012
Discovered: 1 Apr 2014
Fixed: 7 Apr 2014

“... can be used to reveal up to 64k of memory to a connected client or server ...”

Costs:

- >$500 million
- 30,000 X.509 certificates compromised
- 4.5 million patient records compromised
- CRA website shutdown, 900 SINs leaked
- ...

Bugs in OpenSSL

Heartbleed

Severity: 7.5 HIGH
Introduced: 14 Mar 2012
Discovered: 1 Apr 2014
Fixed: 7 Apr 2014

"... can be used to reveal up to 64k of memory to a connected client or server ..."

Costs:

- >$500 million
- 30,000 X.509 certificates compromised
- 4.5 million patient records compromised
- CRA website shutdown, 900 SINs leaked
- ...

? CVE-2016-6309

Severity: 9.8 CRITICAL
Introduced: 22 Sep 2016
Discovered: 23 Sep 2016
Fixed: 26 Sep 2016

"... likely to result in a crash, however it could potentially lead to execution of arbitrary code ..."

Costs:

- minimal

Bugs in OpenSSL

Heartbleed

Severity: 7.5 HIGH

Int…

Di…

Fixed: 7 Apr 2014

“... can be used to reveal up to 64k of memory to a connected client or server ...”

Costs:

- >$500 million
- 30,000 X.509 certificates compromised
- 4.5 million patient records compromised
- CRA website shutdown, 900 SINs leaked
- ...

by honggfuzz
(modern coverage-guided fuzzer)

CVE-2016-6309

Severity: 9.8 CRITICAL

Introduced: 22 Sep 2016

Discovered: 23 Sep 2016

Fixed: 26 Sep 2016

“... likely to result in a crash, however it could potentially lead to execution of arbitrary code ...”

Costs:

- minimal

Bugs in OpenSSL

Heartbleed

Severity: 7.5 HIGH

In

Di

Fixed: 7 Apr. 2014

by honggfuzz
(modern coverage-guided fuzzer)

“... can be used to reveal up to 64k of ... clie

Costs:

- >$500 million
- 30,000 X.509 certificates compromised
- 4.5 million patient records compromised
- CRA website shutdown, 900 SINs leaked
- ...

?

CVE-2016-6309

Severity: 9.8 CRITICAL

Introduced: 22 Sep 2016

Discovered: 23 Sep 2016

Fixed: 26 Sep 2016

“... likely to result in a ... ould ... ecution ...”

- minimal

In this class, we will study the tools & techniques that enabled this rapid discovery

Goal(s) of this Course

Intro to ***research*** in automated testing, bug detection, and program analysis

Schedule for Today

- Introductions
- Class format/logistics
- What is automated testing, bug detection, program analysis?
- Black-box/Random Fuzzing
- TODOs for next time

Schedule for Today

- Introductions
- Class format/logistics
- What is automated testing, bug detection, program analysis?
- Black-box/Random Fuzzing
- TODOs for next time

Introductions

- Name, year, advisor/lab/research interests
- Why are you interested in this class?

Schedule for Today

- Introductions
- Class format/logistics
- What is automated testing, bug detection, program analysis?
- Black-box/Random Fuzzing
- TODOs for next time

Goal(s) of this Course

Intro to ***research*** in automated testing, bug detection, and program analysis

- Become familiar with key techniques in automated testing
  - **Class activities:** paper responses, lecture
- Read and critically evaluate papers in the field
  - **Class activities:** paper responses, discussion
- Assess which problems are well-suited to different automated testing techniques
  - **Class activities:** paper responses, discussion, lecture, assignment
- Assess, design, and conduct experiments of a program analysis/testing tool
  - **Class activities:** paper responses, discussion, assignment, project
- Design and conduct a research project in program analysis/testing
  - **Class activities:** project

Goal(s) of this Course

Intro to ***research*** in automated testing, bug detection, and program analysis

- Become familiar with key techniques in automated testing
  - **Class activities:** paper responses, lecture
- Read and critically evaluate papers in the field
  - **Class activities:** paper responses, discussion
- Assess which problems are well-suited to different automated testing techniques
  - **Class activities:** paper responses, discussion, lecture, assignment
- Assess, design, and conduct experiments of a program analysis/testing tool
  - **Class activities:** paper responses, discussion, assignment, project
- Design and conduct a research project in program analysis/testing
  - **Class activities:** project

Class Format

This class has 3 main components:

- Paper responses
- Assignment
- Course Project

Paper Responses

Before each class (except designated classes), you will read a research paper + post a response on Piazza

Paper responses should summarize the paper + your opinions to help spark discussion in class

In-class discussions of the paper will be led by a discussion lead (student). Goal of the discussion is to deepen the understanding + critical analysis of the subject matter.

Class Format

This class has 3 main components:

- Paper responses (35%)
  - (20%) Responses, due 18 hours before class
  - (10%) In class participation in discussions + Piazza participation
  - (5%) Discussion lead: read other students' responses and prepare to lead discussion in-class
- Assignment
- Course Project

Assignment

You will implement a random + coverage-guided fuzzer in Python

You will evaluate these fuzzers on different benchmarks, and write up a (guided) analysis of the reults

Released **today**, if you want to take a look!

Class Format

This class has 3 main components:

- Paper responses (35%)
  - (20%) Responses, due 18 hours before class
  - (10%) In class participation in discussions + Piazza participation
  - (5%) Discussion lead: read other students' responses and prepare to lead discussion in-class
- Assignment (12.5%)
- Course Project

Project

Open-ended, choose a topic related to automated testing, program analysis, bug detection.

Potential project types (see website for concrete suggestions):

- develop a new tool for testing in a particular domain
- tweaking an existing algorithm: evaluate effect of change
- an extensive re-evaluation of an existing tool
- a reimplementation of an algorithm in a new domain
- or creating a benchmark suite.

You may work in groups, or alone (only recommended if you have a topic closely connected to your research field)

Project

Proposal: describe background of the project, goal + intended deliverables, evaluation plan, timeline, division of work.

Check-in: 1 month in, update on any changes to plan since proposal

Writeup: background of the project, intended goal, what was accomplished, evaluation results, division of work

Presentation: summarize background and key achievements to class

Class Format

This class has 3 main components:

- Paper responses (35%)
  - (20%) Responses, due 18 hours before class
  - (10%) In class participation in discussions + Piazza participation
  - (5%) Discussion lead: read other students' responses and prepare to lead discussion in-class
- Assignment (12.5%)
- Course Project (52.5%)
  - (10%) Proposal: due Fri Oct 14
  - (2.5%) Check-in: due Fri Nov 18
  - (30%) Writeup: due Dec 16
  - (10%) Presentation: week of Dec 5th, length TBD depending on # groups

Attendance + Late Policy

- This class relies on regular in-person attendance of discussions
  - Reading papers alone: ☹
  - Discussing papers with others: ☺
- Paper responses must be submitted on-time
- All other class deadlines (assignment, project) are Fridays at 6pm
- If you anticipate being unable to attend class for some legitimate reason, please inform me ahead of time
  - *Especially* if you are signed up as discussion lead

Academic Honesty

- **Responses:** You may discuss papers with other students, but write your paper responses alone. Do not read other students' responses on Piazza before submitting yours.
- **Assignment:** You may discuss the assignment with other students, but the code + writeup should be your own. Attribute any code from StackOverflow, etc, accordingly.
- **Project:** You may use other people's code as a base for your project; attribute the source of any piece of code outside of the main project. Do not plagiarize any text for your proposal + writeup: it should be written by you and your teammates alone.

Schedule for Today

- Introductions
- Class format/logistics
- What is automated testing, bug detection, program analysis?
- Black-box/Random Fuzzing
- TODOs for next time

Automated Testing ≠ Test Automation

Test Automation

https://www.functionize.com/automated-testing

What is automated testing?

Automated testing refers to any approach that makes it possible to run your tests without human intervention. Traditional testing has been done manually. A human follows a set of steps to check whether things are behaving as expected. By contrast, an **automated test is created once** and then can run any time you need it.

For a long time, developers have automated their unit testing. That is, the tests that check whether a given function is working properly. Then automated testing frameworks like Selenium were developed. These allow modules or entire applications to be tested automatically.

These frameworks allow a test script to interact with your UI, replicating the actions of a user. For instance, they allow you to find a specific button and click it. Or locate a text entry box and fill it out correctly. They also allow you to verify that the test has completed correctly.

https://www.atlassian.com/devops/devops-tools/test-automation

What is test automation?

Test automation is the practice of automatically reviewing and validating a software product, such as a web application, to make sure it meets predefined quality standards for code style, functionality (business logic), and user experience.

Testing practices typically involve the following stages:

- **Unit testing**: validates individual units of code, such as a function, so it works as
- **Integration testing**: ensures several pieces of code can work together without u consequences
- **End-to-end testing**: validates that the application meets the user's expectation:
- **Exploratory testing**: takes an unstructured approach to reviewing numerous are application from the user perspective, to uncover functional or visual issues

The different types of testing are often visualized as a pyramid. As you climb up the the number of tests in each type decreases, and the cost of creating and running te increases.

https://en.wikipedia.org/wiki/Test_automation

Test automation

From Wikipedia, the free encyclopedia

See also: Manual testing

This article includes a list of general references, but **it lacks sufficient corresponding inline citations**. Please help to improve this article by introducing more precise citations. *(February 2009)* (*Learn how and when to remove this template message*)

In software testing, **test automation** is the use of software separate from the software being tested to control the execution of tests and the comparison of actual outcomes with predicted outcomes.[1] Test automation can automate some repetitive but necessary tasks in a formalized testing process already in place, or perform additional testing that would be difficult to do manually. Test automation is critical for continuous delivery and continuous testing.[2]

This is not what we will cover in class

https://www.functionize.com/automated-testing

What is automated testing?

Automated testing refers to any approach that makes it possible to run your tests without human intervention. Traditional testing has been done manually. A human follows a set of steps to check whether things are behaving as expected. By contrast, an **automated test is created once** and then can run any time you need it.

For a long time, developers have automated their unit testing. That is, the tests that check whether a given function is working properly. Then automated testing frameworks like Selenium were developed. These allow modules or entire applications to be tested automatically.

These frameworks allow a test script to interact with your UI, replicating the actions of a user. For instance, they allow you to find a specific button and click it. Or locate a text entry box and fill it out correctly. They also allow you to verify that the test has completed correctly.

https://www.atlassian.com/devops/devops-tools/test-automation

What is test automation?

Test automation is the practice of automatically reviewing and validating a software product, such as a web application, to make sure it meets predefined quality standards for code style, functionality (business logic), and user experience.

Testing practices typically involve the following stages:

- **Unit testing**: validates individual units of code, such as a function, so it works as
- **Integration testing**: ensures several pieces of code can work together without u consequences
- **End-to-end testing**: validates that the application meets the user's expectation:
- **Exploratory testing**: takes an unstructured approach to reviewing numerous are application from the user perspective, to uncover functional or visual issues

The different types of testing are often visualized as a pyramid. As you climb up the the number of tests in each type decreases, and the cost of creating and running te increases.

https://en.wikipedia.org/wiki/Test_automation

Test automation

From Wikipedia, the free encyclopedia

See also: Manual testing

This article includes a list of general references, but **it lacks sufficient corresponding inline citations**. Please help to improve this article by introducing more precise citations. *(February 2009) (Learn how and when to remove this template message)*

In software testing, **test automation** is the use of software separate from the software being tested to control the execution of tests and the comparison of actual outcomes with predicted outcomes.[1] Test automation can automate some repetitive but necessary tasks in a formalized testing process already in place, or perform additional testing that would be difficult to do manually. Test automation is critical for continuous delivery and continuous testing.[2]

Automated Testing

- Test-input generation
  - *Generate test inputs that expose bugs in a program*

- Test case / Test Suite Generation
  - *Generate test suites that expose bugs in a program*

Automated Testing

- Test-input generation
  - *Generate test inputs that expose bugs in a program*

- Test case / Test Suite Generation
  - *Generate test suites that expose bugs in a program*

Randoop, EvoSuite: will cover later in class

Automated Testing

- Test-input generation
  - *Generate test inputs that expose bugs in a program*

Fuzzing, Concolic + Symbolic Execution

- Test case / Test Suite Generation
  - *Generate test suites that expose bugs in a program*

Test-Input Generation

- Assume a program *P* which takes in input *i*

Example Program *P*

```
def cgi_decode(s: str) -> str:
 """Decode the CGI-encoded string `s`: * replace '+' by ' ' * replace "%xx" by the character
with hex number xx. Return the decoded string. Raise `ValueError` for invalid inputs."""
    t = ""
    i = 0
    while i < len(s):
        c = s[i]
        if c == '+’:
            t += ' ‘
        elif c == '%’:
            digit_high, digit_low = s[i + 1], s[i + 2]
            i += 2
            if digit_high in hex_values and digit_low in hex_values:
                v = hex_values[digit_high] * 16 + hex_values[digit_low] t += chr(v)
            else:
                raise ValueError("Invalid encoding")
        else:
            t += c
        i += 1
    return t
```

Example Program *P*

```python
def cgi_decode(s: str) -> str:
 """Decode the CGI-encoded string `s`: * replace '+' by ' ' * replace "%xx" by the character
with hex number xx. Return the decoded string. Raise `ValueError` for invalid inputs."""
    t = ""
    i = 0
    while i < len(s):
        c = s[i]
        if c == '+’:
            t += ' ‘
        elif c == '%’:
            digit_high, digit_low = s[i + 1], s[i + 2]
            i += 2
            if digit_high in hex_values and digit_low in hex_values:
                v = hex_values[digit_high] * 16 + hex_values[digit_low] t += chr(v)
            else:
                raise ValueError("Invalid encoding")
        else:
            t += c
        i += 1
    return t
```

Example Program *P*, Input *i*

Hello%21+World%22

```
def cgi_decode(s: str) -> str:
 """Decode the CGI-encoded string `s`: * replace '+' by ' ' * replace "%xx" by the character
with hex number xx. Return the decoded string. Raise `ValueError` for invalid inputs."""
    t = ""
    i = 0
    while i < len(s):
        c = s[i]
        if c == '+':
            t += ' '
        elif c == '%':
            digit_high, digit_low = s[i + 1], s[i + 2]
            i += 2
            if digit_high in hex_values and digit_low in hex_values:
                v = hex_values[digit_high] * 16 + hex_values[digit_low] t += chr(v)
            else:
                raise ValueError("Invalid encoding")
        else:
            t += c
        i += 1
    return t
```

Example Program *P*, Input *i*

Hello%21+World%22

```
def cgi_decode(s: str) -> str:
 """Decode the CGI-encoded string `s`: * replace '+' by ' ' * replace "%xx" by the character
with hex number xx. Return the decoded string. Raise `ValueError` for invalid inputs."""
    t = ""
    i = 0
    while i < len(s):
        c = s[i]
        if c == '+':
            t += ' '
        elif c == '%':
            digit_high, digit_low = s[i + 1], s[i + 2]
            i += 2
            if digit_high in hex_values and digit_low in hex_values:
                v = hex_values[digit_high] * 16 + hex_values[digit_low] t += chr(v)
            else:
                raise ValueError("Invalid encoding")
        else:
            t += c
        i += 1
    return t
```

Example Program *P*, Input *i*, Result *P(i)*

Hello%21+World%22

```
def cgi_decode(s: str) -> str:
 """Decode the CGI-encoded string `s`: * replace '+' by ' ' * replace "%xx" by the character
with hex number xx. Return the decoded string. Raise `ValueError` for invalid inputs."""
    t = ""
    i = 0
    while i < len(s):
        c = s[i]
        if c == '+':
            t += ' '
        elif c == '%':
            digit_high, digit_low = s[i + 1], s[i + 2]
            i += 2
            if digit_high in hex_values and digit_low in hex_values:
                v = hex_values[digit_high] * 16 + hex_values[digit_low] t += chr(v)
            else:
                raise ValueError("Invalid encoding")
        else:
            t += c
        i += 1
    return t
```

That input exercised the code highlighted green

Returned normally

Example Program *P*, Input *i*, Result *P(i)*

Hello%2V+World%22

```
def cgi_decode(s: str) -> str:
 """Decode the CGI-encoded string `s`: * replace '+' by ' ' * replace "%xx" by the character
with hex number xx. Return the decoded string. Raise `ValueError` for invalid inputs."""
    t = ""
    i = 0
    while i < len(s):
        c = s[i]
        if c == '+':
            t += ' '
        elif c == '%':
            digit_high, digit_low = s[i + 1], s[i + 2]
            i += 2
            if digit_high in hex_values and digit_low in hex_values:
                v = hex_values[digit_high] * 16 + hex_values[digit_low] t += chr(v)
            else:
                raise ValueError("Invalid encoding")
        else:
            t += c
        i += 1
    return t
```

That input exercised the code highlighted green

Returned a ValueError

Test-Input Generation

- Assume a program *P* which takes in input *i*
- Goal of automated Test-Input Generation:
  - Given *P*, generate inputs *i* which expose bugs

Test-Input Generation

- Assume a program *P* which takes in input *i*
- Goal of automated Test-Input Generation:
  - Given *P*, generate inputs *i* which expose bugs… or other interesting behaviors

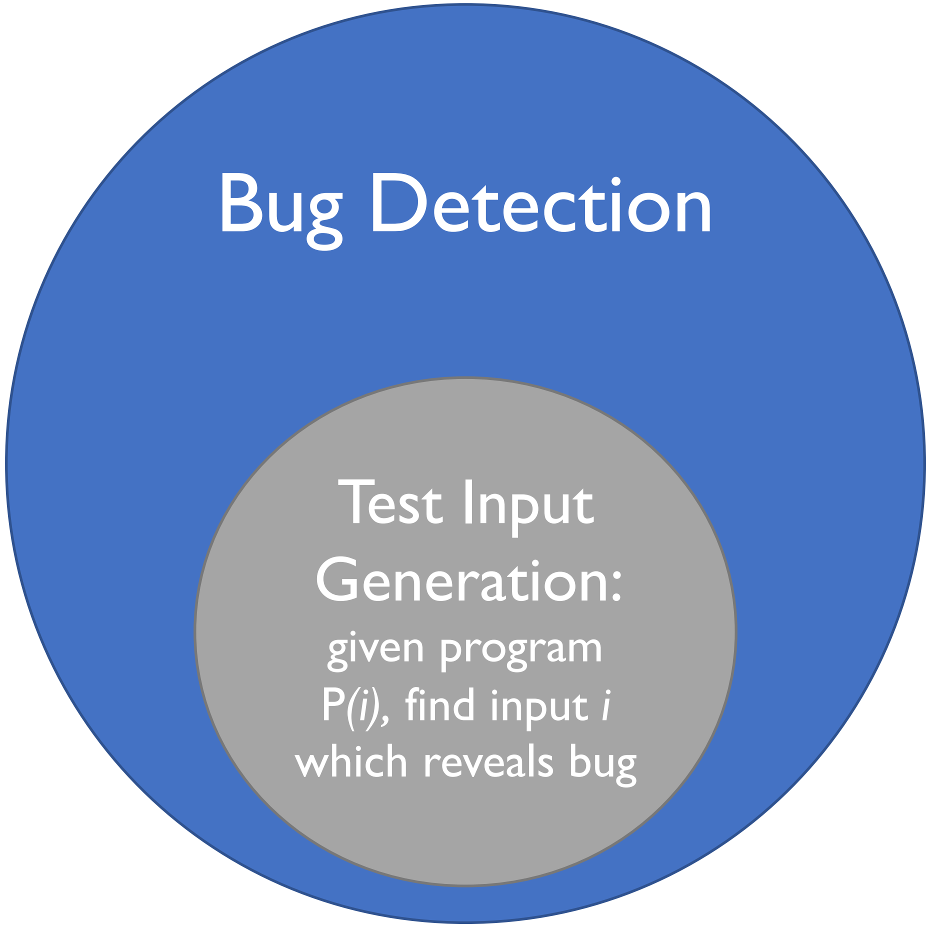
Bug Detection

Broader than Test-Input Generation

Test Input Generation:
given program P(*i*), find input *i* which reveals bug

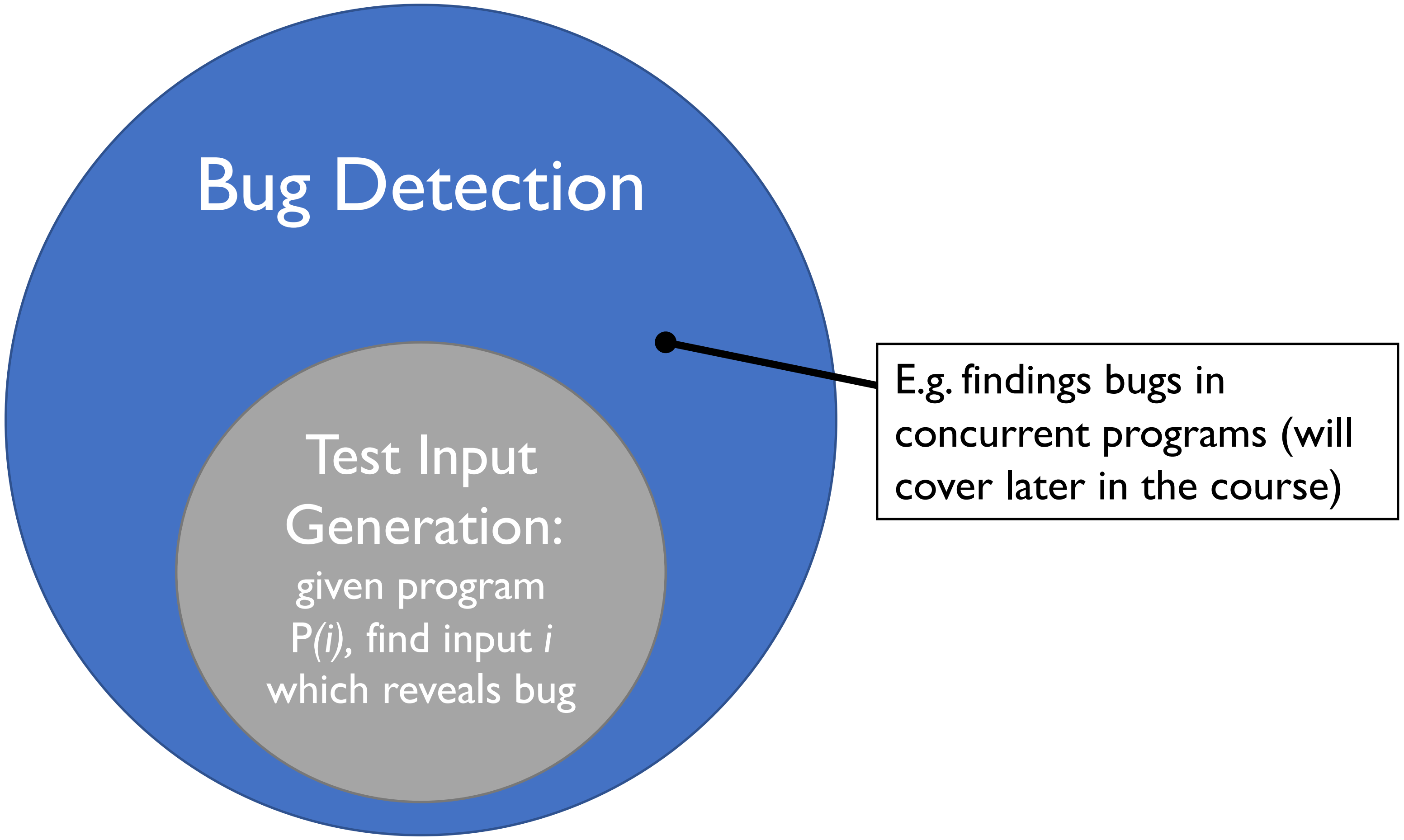
Bug Detection

Broader than Test-Input Generation



Bug Detection

Broader than Test-Input Generation



Program Analysis

Dynamic Analysis

Given a program *P*, and an input *i*, analyze *P* while it executes on input *i*: *analyze(P(i))*

e.g. taint analysis: which parts of the input *i* are used in different parts of the program?

Static Analysis

Analyze a program P independent of input *i*: *analyze(P)*

e.g. data flow analysis, pattern checking in your compiler

Schedule for Today

- Introductions
- Class format/logistics
- What is automated testing, bug detection, program analysis?
- Blackbox/Random Fuzzing
- TODOs for next time

What is Fuzzing/Fuzz Testing?

Aims to solve the test-input generation problem:

"Given program *P* generate inputs *i* which expose bugs or other interesting behaviors "

Fuzzing algorithms are test-input generation algorithms where:

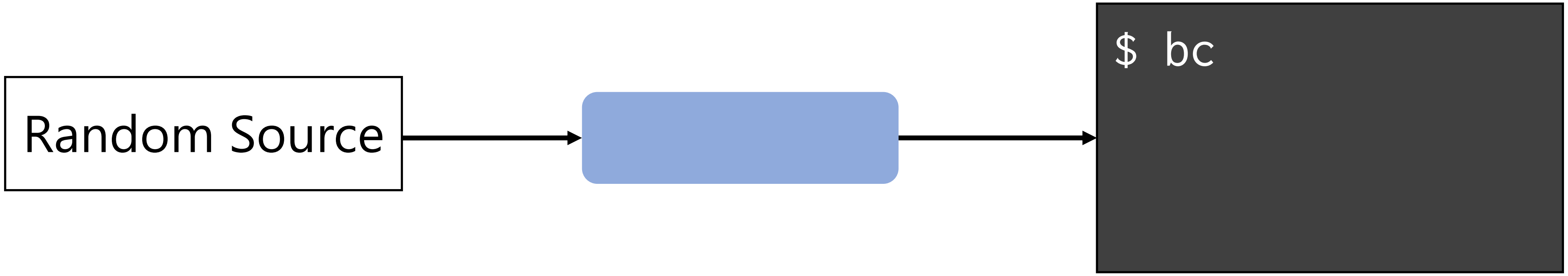
- Fuzzing algorithm has some elements of randomness
- Fuzzing algorithm may use feedback from program execution: *P(i)* or *analyze(P(i))* to guide the generation of the next input

Simplest: Random Fuzzing

Given a a program *P*, generate input *i* randomly.

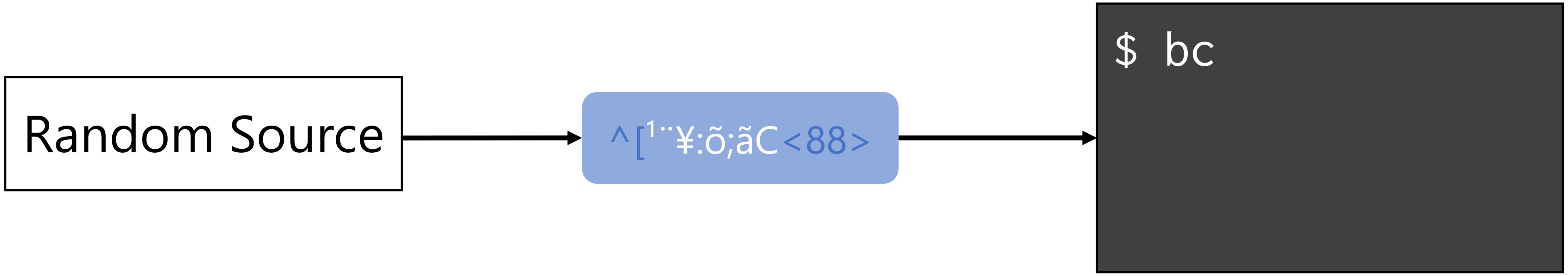
Called "blackbox fuzzing" because it is not using any feedback from the program under test [*P(i)* or *analyze(P(i))*] to guide input generation

Random Fuzzing



B. Miller, L. Fredriksen, B. So. *An Empirical Study of the Reliability of Unix Utilities.* Communications of the ACM, 1990.

Random Fuzzing

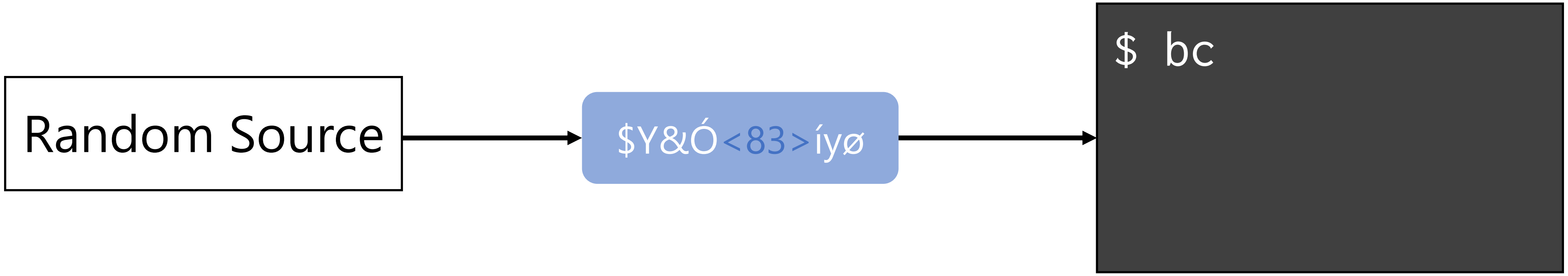


B. Miller, L. Fredriksen, B. So. *An Empirical Study of the Reliability of Unix Utilities.* Communications of the ACM, 1990.

Random Fuzzing

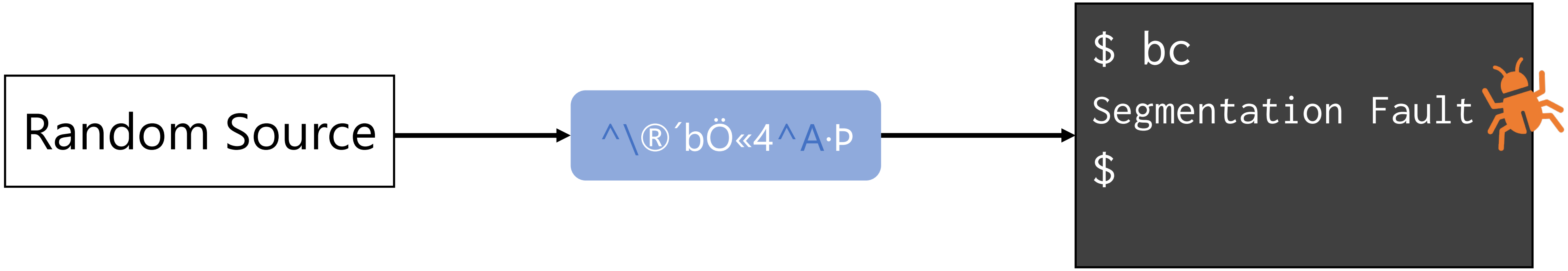
B. Miller, L. Fredriksen, B. So. *An Empirical Study of the Reliability of Unix Utilities.* Communications of the ACM, 1990.

Random Fuzzing



B. Miller, L. Fredriksen, B. So. *An Empirical Study of the Reliability of Unix Utilities.* Communications of the ACM, 1990.

Random Fuzzing



B. Miller, L. Fredriksen, B. So. *An Empirical Study of the Reliability of Unix Utilities.* Communications of the ACM, 1990.

Random Fuzzing

B. Miller, L. Fredriksen, B. So. *An Empirical Study of the Reliability of Unix Utilities.* Communications of the ACM, 1990.

Reading for next time: First "Fuzzing" Paper

Barton P. Miller, Lars Fredriksen and Bryan So

An Empirical Study of the Reliability of UNIX Utilities

When we use basic operating system facilities, such as the kernel and major utility programs, we expect a high degree of reliability. These parts of the system are used frequently and this frequent use implies that the programs are well-tested and working correctly. To make a systematic statement about the correctness of a program, we should probably use some form of formal verification. While the technology for program verification is advancing, it has not yet reached the point where it is easy to apply (or commonly applied) to large systems.

A recent experience led us to believe that, while formal verification of a complete set of operating system utilities was too onerous a task, there was still a need for some form of more complete testing: On a dark and stormy night one of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. The author had to race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash. These programs included a significant number of basic operating system utilities. It is reasonable to expect that basic utilities should not crash ("core dump"); on receiving unusual input, they might exit with minimal error messages, but they should not crash. This experience led us to believe that there might be serious bugs lurking in the systems that we regularly used.

This scenario motivated a systematic test of the utility programs running on various versions of the Unix operating system. The project proceeded in four steps: (1) programs were constructed to generate random characters, and to help test interactive utilities; (2) these programs were used to test a large number of utilities on random input strings to see if they crashed; (3) the strings (or types of strings) that crash these programs were identified; and (4) the causes of the program crashes were identified and the common mistakes that cause these crashes were categorized. As a result of testing almost 90 different utility programs on seven versions of Unix™, we were able to crash more than 24% of these programs. Our testing included versions of Unix that underwent commercial product testing. A byproduct of this project is a list of bug reports (and fixes) for the crashed programs and a set of tools available to the systems community.

There is a rich body of research on program testing and verification. Our approach is not a substitute for a formal verification or testing procedures, but rather an inexpensive mechanism to identify bugs and increase overall system reliability. We are using a coarse notion of correctness in our study. A program is detected as faulty only if it crashs or hangs (loops indefinitely). Our goal is to complement, not replace, existing test procedures.

This type of study is important for several reasons: First, it contributes to the testing community a large list of real bugs. These bugs can provide test cases against which researchers can evaluate more sophisticated testing and verification strategies. Second, one of the bugs that we found was caused by the same programming practice that provided one of the security holes to the Internet worm (the "gets finger" bug) [2,3] We have found additional bugs that might indicate future security holes. Third, some of the crashes were caused by input that might be carelessly typed—some strange and unexpected errors were uncovered by this method of testing. Fourth, we sometimes inadvertently feed programs noisy input (e.g., trying to edit or view an object module). In these cases, we would like some meaningful and predictable response. Fifth, noisy phone lines are a reality, and major utilities (like shells and editors) should not crash because of them. Last, we were interested in the interactions between our random testing and more traditional industrial software testing.

While our testing strategy sounds somewhat naive, its ability to discover fatal program bugs is impressive. If we consider a program to be a complex finite state machine, then our testing strategy can be thought of as a random walk through the state space, searching for undefined states. Similar techniques have been used in areas such as network protocols and CPU cache testing. When testing network protocols, a module can be inserted in the data stream. This module randomly perturbs the packets (either destroying them or modifying them) to test the protocol's error detection and recovery features. Random testing has been used in evaluating complex hardware, such as multiprocessor cache coherence protocols [4]. The state space of the device, when combined with the memory architecture, is large enough that it is difficult to generate systematic tests. In the multiprocessor example, random generation of test cases helped cover a large part of the state space and simplify the generation of cases.

Unix is a trademark of AT&T Bell Laboratories.

Schedule for Today

- Introductions
- Class format/logistics
- What is automated testing, bug detection, program analysis?
- Black-box/Random Fuzzing
- TODOs for next time

TODOs

- Sign up for Piazza
  - *In this course, you will be using Piazza, which is a tool to help facilitate discussions. When creating an account in the tool, you will be asked to provide personally identifying information. Please know you are not required to consent to sharing this personal information with the tool, if you are uncomfortable doing so. If you choose not to provide consent, you may create an account using a nickname and a non-identifying email address, then let your instructor know what alias you are using in the tool.*
- On Piazza: Respond to sign-up for discussion lead
- Paper response for first paper due Sunday 2:30pm
- Assignment due September 23rd at 6pm