Testing

Objectives

Software powers more and more critical components of everyday life, from banking to automobiles to power plants. Software bugs can be a mere inconvenience in some domains, but they can also cause disasters in others, including loss of life. This is where testing comes in: we test software to gain confidence that it does what we want it to.

In this module, you will learn:

- What kinds of tests there are
- How to design automated tests
- How to evaluate tests
- How to test code with external dependencies
- How testing can help development
- What problems can arise due to tests

Testing a program consists of executing the program with a given set of inputs and observing its behavior. As we've seen in SwEng already, testing is necessary, and it's about finding bugs, not about proving their absence.

Test automation entails running several tests in an automated fashion, such as every night or every time a major change is made to the program. Such automation requires one or more test cases, each consisting of specific inputs to the program, and an automated means of validating the outcome, often called a test oracle. Test cases can be written in a black-box manner (where the test developer chooses scenarios solely based on knowledge of what the program is supposed to do) or a white-box manner (where internal knowledge of the program source code supplements the external knowledge in choosing test scenarios). There exist many types of testing (e.g., unit, feature, functional, system, regression) that fulfill various goals, and we'll cover these below. Testing is complementary to other non-dynamic methods for checking program correctness, such as visual code inspection or static program analysis.

But, to talk about testing, we must consider what code *should* do in the first place.

There are some basics that everyone can agree on: not crash, not cause damage to humans or property, not lose data, and other “nots”. But one cannot do any of those by not doing anything! Clearly, we need a way to express what a program should do.

That is where *requirements* come in: they are simple descriptions of what some piece of code should do. They can be as simple as “that button should be red”, or as complex as “data should be persisted even in 15 different failure cases”. They do not need to be formal, and can even be about human feelings, such as “users should enjoy using the software”.

Requirements are often not explicit; a software engineer must derive some of them from context and ask customers about the rest. However, they always exist! If a developer writes software without any requirements, they have no idea what they are doing.

A more formal version of requirements is *specifications*. Specifications are unambiguous requirements, i.e. two people must be able to agree on what exactly they mean. One can then verify that code meets a specification, in a formal or informal way.

Formal verification consists of a proof that the code is correct. The proof is either correct or incorrect; there is no probability involved. This is the ideal form of verification. However, it is a hard and time-consuming process. Developers must annotate their code with proofs, often resulting in 10-20 lines of proof per line of code.

An alternative form of verification is *testing*: checking the behavior of software in a set of well-known cases. Testing gives developers confidence in code, but it is not a proof! As Edsger W. Dijkstra once said, “Program testing can be used to show the presence of bugs, but never to show their absence!”.

This lecture is about testing. First, why bother testing if it does not offer guarantees? Second, how to write tests? Third, what kind of code should we test? Fourth, when should one write tests in the development process? Finally, how to evaluate tests, and how does this map to people’s needs?

Why test?

We have just seen that testing, unlike formal verification, does not offer strong guarantees against bugs. Testing can only show the presence of bugs, but it is always possible that we forgot to test a specific corner case of the program’s behavior, in which the program does not behave as per its specification.

You have already written a significant amount of code in past courses and projects. Did you do much testing beyond “it seems to work”? This is fine for a class project, in which failure means a lower grade.

Imagine doing this for a university’s student management system: if the system crashes when a student registers for classes, they will not be happy! Clearly a developer of this kind of system needs some confidence that this case will not happen.

Now think about a bank: if the system crashes during a money transfer, where does the money go? The developers of such a system need a lot of confidence in it, yet it may already be too complex for formal verification.

As a final example, consider the Therac-25, a radiation therapy machine. A programming error caused it to give radiation doses hundreds of times greater than it was supposed to, leading to serious injuries and death.

These examples should convince you that even if 100% guarantees are not achievable, testing is useful.

Manual testing

The easiest way to do tests is *manually*: write code, run the code, check that the code does what you think it should. But this approach has many disadvantages:

- You may forget to run a test, and believe the tests all pass even if one does not
- You may make a mistake when running tests, and believe the code is correct even if it is not
- It takes a lot of effort to do, since it is a series of manual steps after every code change
- You can only test code that already exists, and must test the entire application at once

Automated testing

These disadvantages are the reason automated testing was born. The programmer writes not only code but also test code, then runs the tests. This implies:

- The computer will not forget to run a test
- The computer will execute the test code as it is, without sometimes making mistakes
- Running tests takes no effort beyond pushing a button
- You can test any code: a single function, a module, the entire application…

There are also some disadvantages. Writing test code is a high fixed cost, though it should pay for itself since running them is so easy. The precision of specifications needs to be much higher for automated than manual tests, since a computer cannot perform human checks such as "this data seems OK", though this also means the computer will not make subjective mistakes.

Tests in Java with JUnit and Hamcrest

The most common Java testing framework is JUnit:

```
@Test
void onePlusOneIsTwo() {
  assertEquals(1 + 1, 2);
}
```

This looks trivial, but unfortunately it is incorrect!

The JUnit documentation tells us that the first parameter is the expected value, and the second parameter is the actual value. Inverting them will make the error messages that appear when tests fail very confusing.

To avoid this problem, and to make tests easier to write in general, one can use Hamcrest:

```
@Test
void onePlusOneIsTwo() {
  assertThat(1 + 1, is(2));
}
```

This is much easier to read and does not allow for mistakes! Hamcrest has all kinds of "matchers" to test the shape of a value, such as a collection:

```
@Test
void emptyOrOneTwoThree() {
  List<Integer> values = ...;
  assertThat(values,
             either(empty())
             .or(contains(1, 2, 3)));
}
```

The error messages are much nicer than the standard JUnit methods, too:

```
"Expected: (an empty collection or iterable containing [<1>, <2>, <3>]) but: was <[42]>"
```

You can easily run JUnit tests from IDEs such as IntelliJ and Eclipse, or from the command line with build tools such as Gradle.

Writing good tests

A good pattern to write tests is the three As: *Arrange, Act, Assert*. First, arrange the situation: create whichever objects your test needs. Then, act: call the methods that the test is about. Finally, assert: check that the outcome is correct according to the specification.

However, make sure each test does not do too much. Imagine putting everything in one test: if it fails, how do you know which part of the code is incorrect? Instead, you should have one test per concept you are testing. This way, you can tell which parts of your code work based on which tests pass.

It is important to remember that test code is code, just like non-test code! It will go in the same repository as the rest of the code, you should have the same standards when writing it, and you should expect the same level of quality from it. Tests are an integral part of your software, since they will need to evolve along with the code they are testing.

Naming is also an important part of writing tests, just like when writing other code. Consider the following four names:

`birthDateIsPrintedCorrectly`

`shareFeatureWorks`

`nameCanIncludeThaiCharacters`

`canParseNumbersAtLeastOneThousand`

The first two are not good test names. “Correctly” does not mean anything in this context – if you asked 10 people how to print a birth date, you may get 11 opinions. Similarly, “Works” is too abstract, there is no universal definition of what a shared feature should do. The last two names are much better: they clearly describe what they are testing.

To share code in tests, instead of copy/pasting, make use of JUnit’s `@BeforeAll`/`@AfterAll` and `@BeforeEach`/`@AfterEach` annotations. The first couple allows you to run global setup and teardown for the class in which they are, which JUnit will run exactly once, while the second couple allows you to run setup and teardown once per test.

What to test?

Now that you know how to write tests, what code should you test?

You can test many levels of software, just like you can write requirements or specifications at many levels: functions, modules, user interfaces…

Low-level tests are much easier to write since they deal with less code and are thus less complex. But high-level tests give you more confidence that the software is correct. Writing tests for all low-level functions does not tell you whether the application composes these functions correctly.

Pure functions

Consider the following code:

```
int sum(int... ints) {
  int sum = 0;
  for (int i : ints) {
      sum += i;
  }
  return sum;
}
```

You can test this simple function by giving it arguments and checking the return value, like so:

```
@Test
void sumOfOneToThreeIsSix() {
  assertThat(sum(1, 2, 3), is(6));
}
```

There is nothing else to it – this is how you write a test for a function.

Modules

Consider the following method, with its documentation:

```
/** Posts the given text
    as the current user
    to the forum. */
void postMessage(String text);
```

There is already one clear difference with the earlier case: this method does not return anything! Thus, we cannot assert anything about its return value, since it does not have one.

Furthermore, the documentation refers to “the current user” and “the forum”, but the method does not take any user- or forum-related parameters.

As a good software engineer, you will ask a colleague or a customer, who will inform you that this method runs in a phone application, gets “the current user” from the phone and communicates to “the forum” over the Internet.

To be able to test this method, let us make the dependencies explicit:

```
class User { ... }

interface AuthService {
  User getCurrentUser();
}

interface HttpClient {
  String post(String url, String body);
}

void postMessage(String text, AuthService auth, HttpClient client);
```

Now the role of the "postMessage" method is much clearer, since it no longer hides authentication and networking internally.

We can now write tests for it, such as a test for what happens when there is no current user:

```
AuthService auth = new AuthService() {
  @Override
  public User getCurrentUser() {
    return null;
  }
};

HttpClient client = ...;

assertThrows(
  IllegalStateException.class,
  () -> postMessage("Hi", auth, client)
);
```

This test is short, and importantly does not need to run on a phone!

Similarly, we can create a fake HTTP client that ensures the code calls it in the right way:

```
HttpClient client = new HttpClient() {
  @Override
  public String post(String url, String body) {
    assertThat(body, is("MSG: Hi"));
    return "OK";
  }
};
```

We can now test "postMessage" even without an Internet connection! This implies we can test all kinds of edge cases, such as "no Internet connection available" or "the forum replies with an error".

Having to change code to test it may seem like an annoyance, or even a reason not to write tests. However, the changes we have just made are not just an improvement for testing: they make the code more modular and more maintainable! There was no reason for a message-posting function to be aware of the phone operating system's authentication service, or even of the existence of a phone.

Similarly, this kind of changes forces you to think about how clean your architecture is: if your tests become a mess because you have to create a ton of fake dependencies before you can do anything, it is an indication that your modules are too closely tied together.

End-to-end testing

To make sure a piece of software works when users run it, you can write end-to-end tests: tests that treat the software as a “black box”, without any knowledge of the internals.

For instance, you can test a command-line app by calling its “main” method, then checking the return code as well as any side-effects on the disk or network.

This is harder to do than a lower-level test, because it needs a testing harness that isolates the app, sets up a fake network, and so on. But the confidence you get from a successful end-to-end test is much higher than from any other test, since it simulates a real user using your app.

Stress testing

An extension of end-to-end testing is stress testing: pushing a system to its limits. For instance, what happens if you open a 4 TB file with an app? Can a web server handle 10,000 concurrent clients?

This is important for software that has more than a few users because there are many reasons why a piece of software could meet its specification when a single user is using it but not do so when there are too many. For instance, if the software is too slow, requests can queue up until the requests queue is full, at which point the software will drop the requests.

A classic example is a university course registration system: many students will try to register as soon as they can, to get into limited-attendance classes. Thus, the system must be ready to handle a large influx of users, even if most days there are very few concurrent users.

When to test?

Consider a simple model of the product lifecycle: There are two steps, developing the product and releasing it to users.

This leaves three times for testing: before developing, between developing and releasing, and after releasing. All three can be useful.

Testing after development

The simplest time to test is after writing code. Write tests, then fix the code until the tests pass. This is easy because the code is ready, thus there will be no need to rewrite the tests due to code changes. It also allows you to choose when to stop: you can write one test, two tests, twenty tests… until you go on to writing the next feature.

However, this approach has disadvantages. Because you have just written the implementation, you risk writing tests that are a copy-paste of the implementation. If you forgot to handle an edge case in the implementation, odds are you will not think of it in the tests either. It is also too late to fix the design of the code if you realize the current design cannot cleanly handle some requirements, since you have already written all the code. Finally, having the choice of when to stop means you can choose to write zero tests; the customers are waiting for the next feature!

Test-driven development

Instead of writing tests after the code, you can write tests before the code, letting them drive the development. This is Test-Driven Development, or *TDD* for short. You first write tests, then code, then you fix the code until the tests pass.

TDD has many advantages. The first one is that it forces you to think about the requirements before coding, since you must translate them to code. It also means you need less debugging: since you write code little by little and run the tests often, you can easily tell which part of the code causes a bug: the one you just wrote! The instant feedback you get from the tests also gives you an emotional boost whenever a test passes, which is welcome during a coding session.

However, TDD also implies a higher time investment than testing after development, since you must start by writing tests. This is a waste of time if you end up radically changing the design of the code when you get to writing code, and thus you should not use TDD if you are just experimenting with design ideas.

Let us now walk through a TDD example step by step. You are a software engineer developing an application for a bank. Your first task is to implement money withdrawal from an account.

The bank tells you that “users can withdraw money from their bank account, but only as much as they have”. This leaves you with a question, which you ask the bank: “can a bank account have a negative balance?”. The bank answers “no”, that is not possible.

Your first step is now to add an “account” class, without any members for now:

```
class Account {
}
```

You then write the first test, which ensures one can withdraw nothing from an account:

```
@Test
void canWithdrawNothing() {
  Account account = new Account(100);
  assertThat(account.withdraw(0), is(0));
}
```

The highlighted sections do not actually exist in the code, so you add them now that you know what they should look like:

```
class Account {
  public Account(int balance) { }
  public int withdraw(int amount) { return -1; }
}
```

Importantly, you do not actually write the code for these members! The only code in there is what is necessary to make the code compile, i.e. returning values in methods that should return something.

You then write another test:

```java
@Test
void cannotInitializeWithNegativeBalance() {
  assertThrows(IllegalArgumentException.class,
               () -> new Account(-1));
}
```

This test does not need any new code in the account class, so you go on to the next test:

```java
@Test
void canWithdrawLessThanBalance() {
  Account account = new Account(100);
  assertThat(account.withdraw(10), is(10));
  assertThat(account.balance(), is(90));
}
```

Writing this test makes you realize you need the highlighted “balance” method, so you add it:

```java
class Account {
  public int balance() { return -1; }
  public Account(int balance) { }
  public int withdraw(int amount) { return -1; }
}
```

You add one last test, for partial withdrawals:

```java
@Test
void partialWithdrawIfBalanceTooLow() {
  Account account = new Account(10);
  assertThat(account.withdraw(20), is(10));
  assertThat(account.balance(), is(0));
}
```

Now you can run the tests… and see them all fail! This is normal, since you did not actually implement the account class. You can now implement it and run the tests every time you make a change until they all pass.

Finally, you go back to your customer, the bank, and ask what is next. They give you another requirement they had forgotten about: the bank can block accounts and withdrawing from a blocked account has no effect. You can now translate this requirement into tests, adding code as needed to make the tests compile, then implement the code. Once you finish, you will go back to asking for requirements, and so on until your application meets all the requirements.

Regression testing

Let us now consider writing tests after releasing software. This seems counter-intuitive; why would you still need tests when your software is already in users’ hands? The reason is simple: your software will have bugs. No matter how careful you are, there will be bugs. Thus, you must have a process in place to handle bug reports.

One effective way to do so is to first reproduce the bug, then fix the bug, then confirm the fix works. Reproducing the bug is necessary for two reasons: one, users may make mistakes and think they found a bug when they did not; and two, you cannot confirm that you fixed the bug unless you know how to trigger it.

To apply this in practice, you can use tests! This is **regression testing**: write a test that fails, which proves the bug exists, then fix the code, then run the test. If the test passes, you have fixed the bug!

It is important to write the test first and make sure it fails. If you only write it after “fixing” the code, you cannot be sure that you reproduced the bug, since the test might have passed even before your fix!

Evaluating tests

There are two main criteria to evaluate the tests you write: coverage, which evaluates how much code you test, and performance, which evaluates how productive you can be while using the tests.

Coverage

The idea of coverage is simple: divide “amount of code executed during testing” by “total amount of code”.

But what does “code” mean in this context? Lines? Statements? Something else? Let us investigate with an example method:

```
int getPriority(User user) {
  if (user == null) throw ...;
  int priority = 100;
  if (user.isInfluential()) priority += 100;
  if (user.hasUnpaidBills()) priority /= 2;
  return priority;
}
```

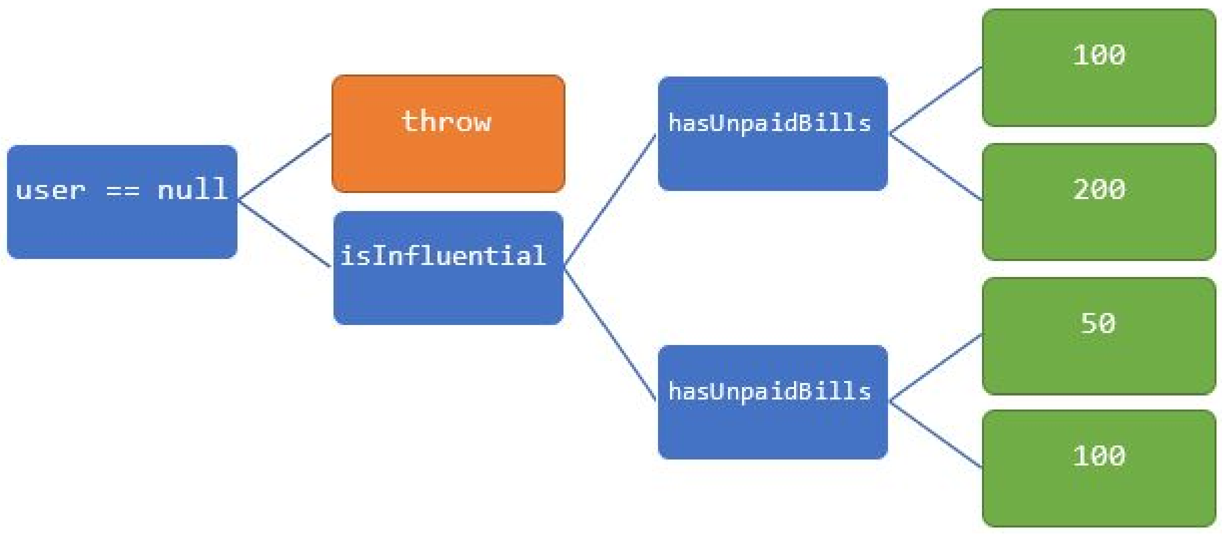
This method is for a shop that wants to prioritize influential customers and customers who pay their bills on time.

Let us write a test for this method, using as input a user that is both influential and with unpaid bills. The *line coverage* is 100%! This is clearly wrong; even in the first line of the method you can tell that there should be at least two tests, one for a null user and one for a non-null one. In fact, if you had written the “throw” statement on a new line, which would not change the semantics of the program, the line coverage would be less than 100%. Clearly, we cannot use a metric that depends on code formatting.

Instead, how about *statement coverage*? The only statement left uncovered by the test we just wrote is the “throw”. This is better, since it is not 100% for our lone test, but still feels wrong. There are two “if” conditions testing the user’s attributes, but somehow a single test covers most of the code?

Time to introduce another form of coverage: *branch coverage*. The idea is that each branch’s condition can evaluate either to “false” or to “true” in one execution. Branch coverage therefore considers the branch evaluations: to cover 100% of the code, all branches must have evaluated once to “false” and once to “true”. Our lone test with an influential user who has unpaid bills now only covers half of the outcomes: the first branch is false, and the last two are true. We must add two more tests to reach 100%: one with a null user, and one with a user that is neither influential nor with unpaid bills.

Is this good enough? To answer this, we must take a step back and analyze this method at a higher level. The “paths” that execution can take in the code look like this:



There is one error case, and four success cases. Note that even though two of the success cases return the same value, execution reaches them through different paths.

Recall that branch coverage gave us a 100% score with three tests, even though there are five paths!

This is where *path coverage* comes in: it tests how many paths the tests go through. With three tests exercising one different path each, we thus get 3/5. To get 100% path coverage, we must add two more tests: one with a user who is influential and does not have unpaid bills, and one with a user who is not influential but has unpaid bills.

But path coverage is not a panacea. Consider the following loop, which asks the user for an input of less than 10 characters and keeps asking until the user supplies one:

```
while (true) {
  String input = getUserInput();
  if (input.length() < 10) break;
  tellUser("Less than 10 chars please");
}
```

How many paths are there in this program? The user could supply a good input the first time. Or the second time. Or the third time. Or the millionth time. Each of those is a path, and thus the number of paths is infinite! Since you can only write a finite number of tests, your path coverage will be 0% no matter how many tests you write.

Consider this different example:

```
if (...) { ... }
if (...) { ... }
if (...) { ... }
if (...) { ... }
if (...) { ... }
```

How many paths are there in this program? The first “if” causes two paths. Each of those paths then branches on the second “if”, for a total of four paths. The next “if” doubles the number of paths again, and so on. These 5 lines of code cause 32 paths! Large pieces of software have thousands of branches, thus the number of paths is so large you could not cover it in your entire life even if you spent all your time writing tests.

Thus, there is a tradeoff in coverage: statement coverage is more feasible but gives less confidence, while path coverage is hard or even infeasible but gives much more confidence. Branch coverage is a midpoint between the two.

It is also important to note that coverage by itself does not guarantee anything. Consider this test:

```
@Test
void coverCode() {
  getPriority(new User(...));
  getPriority(new User(...));
  getPriority(new User(...));
}
```

This test runs the method three times, which can cover a lot of code, but it does not check the return value at all! The only guarantee provided by the test is that the code does not crash with these inputs, since a crash would cause a test failure.

Test performance

The other metric with which you can evaluate tests is how fast they run. This is important because if they take too long to run, you will completely forget the context of what you were doing. Not only will you lose time waiting for the tests to finish, you will lose time becoming productive again by remembering what it is that you were doing!

There are three components to performance:

- How fast each test runs individually
- How fast one can run all tests
- The number of tests you run

The usual cause of slow tests is timeouts: waiting for something to happen without knowing how long it will take. For instance, an application that makes a request to the Internet could wait for a while before getting the response. A naïve way to test it is to wait for 10 seconds after making the request, because it is likely that the application will have received the response in that time. But if the app receives the response in 1 second, that means the test will waste 9 seconds doing nothing! Instead, the application should allow the tests to supply a callback that it will call whenever it receives the response.

You can run tests faster if your machine can run multiple tests at once, using more than one CPU core. However, this means your tests must not depend on each other! If one test expects another test to have performed some operation, running them in parallel will break it. Even if you do not run tests in parallel, you should avoid hidden dependencies since they make the tests hard to understand.

Finally, you do not have to run all tests all the time. You can mark some tests as “fast” tests that you always want to run after changing the code, e.g. using JUnit’s `@Tag` annotation. Then you can run only those tests most of the time, and run the other tests only before merging a feature into the main branch of your repository, since it is likely that if all fast/low-level tests pass the slow/high-level tests will pass as well.

Validation: are you building the right thing?

Now that we have talked about testing, let us take a step back and think about what code is for. Do we write code to pass tests? To run fast? To look good so our teammates will compliment our clean code? No! We write code to help humans. Code helps people use machinery, analyze data, automate tasks, and so on. Thus, no matter how “correct” a piece of code is, it is useless if it does not satisfy its human users.

Let us walk through a tragic example of “correct” code that did not work as intended: the Boeing Maneuvering Characteristics Augmentation System, or MCAS for short. This was a system in Boeing’s 737 MAX plane, which makes sure the plane cannot stall. You do not need to know what stalling is beyond the fact that it is bad for a plane to stall. Engineers implemented MCAS as specified, and yet it was the cause of two crashes that each killed over 150 people. Why?

First, MCAS was not what users wanted: after the first test flight, the test pilot wanted Boeing to change the hardware design of the plane so it would fly more smoothly, but Boeing decided to implement a software fix instead.

Second, MCAS was unknown to users: Boeing voluntarily chose to not include it in the training manual for pilots, believing it was so fundamental to the plane’s design that pilots should never notice it.

Third, MCAS surprised users: Boeing decided that MCAS should be able to override everything else, including humans. This is different from earlier planes in which the pilot could override the machine.

Finally, MCAS did not meet users’ standards: it only used one sensor for its input data, going against the airplane industry’s standard of redundancy since a plane must not crash because of a single faulty piece of hardware.

These errors resulted in a system that was correct according to Boeing’s specification but clearly not what any user wanted.

How can you avoid this? One way is to change your development process to include more user feedback, which the next lecture covers.

Another way is to perform acceptance tests, in which you let a user use the system and tell you whether it is what they wanted. Even if your code is correct according to the requirements, you may discover that there was a misunderstanding and the requirements you wrote down are not what the user wanted.

A/B testing

One way to discover what users want is A/B testing: giving some users version “A” of your software and some users version “B”, then giving versions a score based on some metric and keeping the one with the highest score.

For instance, do users buy your product more if the “Buy!” button is green or red? This may seem absurd, but in real life it does make a difference!

Most of the time you do not want users to be aware that they are being A/B tested, since this may alter their behavior. You must also be careful with the statistics involved to avoid making incorrect conclusions because of statistical issues such as sample sizes.

Testing real-world code

The examples given above are short and self-contained, but real-world code is rarely like this. Consider a cooking application that uses a database to store recipes. Most modules in the application will deal with the database in one way or another, as it is central to the application. The main risk in this context is that most of the codebase will end up implicitly depending on the database module, making it hard to test in isolation.

A common pattern in “easy to use” libraries is to provide a global instance of the main library object, such as a static method returning a database instance:

```
public class SomeDatabase {
    public static Database getDatabase() { ... }
}
```

If your code uses this method directly, it depends on the database and won't be testable in isolation.

Instead, create your own interface to access the database:

```
public interface MyDatabase {
    // the methods your code needs
}
```

You can then change your modules to take an instance of this interface as a constructor parameter, for instance. This allows you to implement the interface using the real operations for production code, and to use fake objects for tests.

In some frameworks, such as Android, it is hard to use constructor parameters because the framework creates your modules for you and does not allow you to add parameters to constructors. In this case, you can create a static method that will return an instance of the production implementation by default, with the possibility to override this in tests:

```
public class MyDatabaseFactory {
    private static MyDatabase database = new MyRealDatabase();

    public static MyDatabase getDatabase() {
        return database;
    }

    // called from test initialization only
    public static void setDatabase(MyDatabase database) {
        MyDatabaseFactory.database = database;
    }
}
```

This leads to less readable code than the constructor option, since the dependency is now implicit, but sometimes there is no other choice.

You may now be thinking, *wait a minute, does that mean I have to write an interface method for each method I want to use in the real library? Isn't this a lot of "boilerplate" code?*

The answer is more nuanced: yes, you will need to write some boilerplate code, but it is also a great opportunity to customize the library's interface for your needs. For instance, if you always call two library methods together, there is no reason to expose both on your custom interface: write a single call that corresponds to what your code expects instead. If you always end up converting from the library's data structures to some other structures that are more adapted to your code, do that inside the implementation as well, exposing a clean interface.

Note that this does not mean the production implementation of your interface will never be tested: that's what end-to-end tests are for.

Automated test generation

The outcome of running tests is measured in various ways. For example, counting how many tests succeed vs. fail is often a proxy metric for the program's code quality. Another example is test coverage, which measures the rigorousness of testing. The fundamental challenge in thoroughly testing a program is that the number of possible inputs is large. For example, a program that takes four 64-bit integers and adds them up: there exist 2^256 different combinations of integers that could be provided to this program. In contrast, scientists estimate that the observable Universe has on the order of 2^240 atoms. A naive approach of trying all inputs one by one would therefore not complete, so one must be clever about picking test inputs.

Meet *test generation* – producing “interesting” inputs to test software, in the hopes of doing more thorough testing more quickly. There is an inherent trade-off between how long it takes to choose the inputs vs. how long it takes to run the corresponding test and measure outcomes. For example, if executing a program takes a long time, it makes sense to spend time on smartly choosing inputs, so as to minimize the number of times the program must execute during testing. However, if running the program is quick, then taking a long time to choose inputs can be detrimental compared to running the program many times with many inputs.

A good way to reduce the time spent choosing test inputs is to automate the process. There is also the human angle: writing tests is like writing code, but less fun. This gives rise to the field of *automated test generation*, which is the latest novelty to hit the field of software engineering when it comes to testing.

The simplest form of automated test generation is to select program inputs at random. This is called "fuzzing" or "fuzz testing". This was first talked about in an article by Miller et al. published in 1990. The approach was surprisingly effective at the time: by automatically trying random inputs, the authors were able to crash 25-33% of the utility programs on any version of UNIX that was tested. The causes of these crashes were related to many of the things we learn about in SwEng: incorrect input validation, not checking return codes, bad error/exception handlers, etc.

A more evolved form of fuzzing is to use various input probability distributions, either uniform or biased towards some specific values believed to lead to interesting corner cases, like 0, −1 or MAXINT for integer input values.

This approach presents several significant advantages:

- Generating the test inputs is blazing fast, since there is no "thinking" involved
- The approach is completely black-box, which makes it easy to use and widely applicable to pretty much any software

It also has serious drawbacks:

- Many inputs are invalid, so they get rejected easily by the program, and are thus a waste of time: producing those inputs serves no real testing purpose.
- Many inputs just exercise the same path exercised by previous tests, which is again a waste of time. Later on we will talk about gray-box and white-box approaches that address this issue head-on.
- You end up testing primarily the program's input checking code, i.e., does the program correctly reject invalid inputs or not. This is because the probability of generating at random a valid input is quite low. The problem with this is that the test inputs do not penetrate deeper into the software, they do not pass the first layer, and therefore the rest of the software remains untested. Feedback-driven fuzzing, discussed below, attempts to address this issue.
- Can at best check for shallow properties, such as whether the program crashes in an unexpected way or not; you can't check if it computes the right thing or takes the right action, unless you spend more time on the testing framework itself.

Key to the effectiveness of fuzzing is test *quantity*, i.e., the ability to generate and try out new inputs at high rate. Large-scale fuzzing efforts (such as ClusterFuzz that tests e.g. the Chromium web browser) test software round the clock, using as many machines as are available. The number of bugs found is limited by the number of CPU resources given to the fuzzer—intuition suggests that the more tests the fuzzer gets to run, the more likely it is to find bugs.

As a simple back-of-the-envelope exercise, take a small program with a bug (e.g., from one of the past SwEng exams), and then calculate the probability of a dumb fuzzer finding an input that hits that particular bug. Once you've done this, play with adding/removing code from the program, and see how this affects the probability. The higher the probability, the sooner you can expect a dumb fuzzer to find the bug. As part of this exercise, you could also estimate how long it would take to find the bug: estimate (i.e., compute the expected value of) how many executions of the program are needed to hit the bug, and multiply that by the average time it takes to run the program once.

The other key ingredient to fuzzing effectiveness is test *quality*. First, testing many random inputs in a blackbox fashion can at best discover shallow bugs, as mentioned earlier, whereas picking inputs smartly can penetrate deeper into the program code and reduce the number of executions needed to find a bug.

An important part of any test is detecting whether it succeeds or not. The main goal of fuzzing is to identify program crashes. If you can't accurately determine if a program has crashed, then you can't determine whether the test case triggers a bug or not. There are at least three good ways to do this:

- Rely on the runtime of a managed language (e.g., look for a stacktrace from a Java program) or attach a debugger to the program. Keep in mind though that attaching a debugger to your program may slow it down by a lot.
- An alternative to attaching with a debugger is to simply monitor its exit code: does it stop with a normal termination code after executing, or does it encounter a segmentation fault, or a division by zero, or …? This avoids the debugger-induced slowdown, but the diagnosis of the crash is less clear.
- If you're testing a program that is not under your full control, and you cannot monitor its exit code, you can use timeouts: if the program normally responds to your input test cases within a certain amount of time, but for a particular input it does not respond by the end of some fixed amount of time, then you can assume that the program has either crashed or frozen/hung.

Whichever method you use, the program should be restarted whenever it crashes or becomes unresponsive, in order to allow fuzzing to continue. It's important that you have clean state for each new test.

As you might intuit by now, fuzzing can be powerful by providing good results with little effort: just let the fuzzer run for hours, days, or months and come back later to see the list of bugs it found. It can find bugs that you would completely miss otherwise, and can give you a picture of how robust the code is to adversarial inputs. Of course, fuzzing won't find all bugs, it can only discover those that lead to a crash. Plus, it's unlikely to find bugs that require complex conditions to occur (e.g., concurrency bugs). If you do find a bug, debugging it can be quite difficult: the fuzzer can give the input that triggers the crash, but little more.

An example of a simple fuzzer that can be used on pretty much any platform is Radamsa. It's goal is to "just work" for a variety of input types and contains a number of different fuzzing algorithms (which go beyond just dumb fuzzing). It has been used to find many bugs which have resulted in dozens of CVEs.

There are various ways to improve the quality of generated inputs to mitigate these drawbacks:

- Leverage structural knowledge of what a *valid* inputs look like
- Leverage feedback from prior executions to steer input generation toward those inputs that are more likely to uncover bugs

Generation-based fuzzing

This is the first idea in "smart fuzzing": use knowledge of inputs to generate only (or mostly) valid inputs. For example, you can split a protocol or file format into chunks, which then the fuzzer can combine in various ways and in a valid order to produce valid input files or input messages (or network packets). You can randomly fuzz these chunks independently to create inputs that preserve their overall structure, but contain inconsistent data within them. The granularity of these chunks and the intelligence with which they're constructed define the level of intelligence of the fuzzer.

SPIKE is an example of a fuzzer in this category. It is really a network protocol fuzzer creator, giving users an API with which you can encode the specifics of a network protocol into a C++-based input generator. The resulting tool can now construct fuzzed messages that can be sent to a network service to test it. It is designed to run on Linux.

Generation-based fuzzing has distinct advantages:

- It can penetrate deeper into a protocol stack or a file processing program, because the valid sequences of inputs pass the initial parsing stage and exercise the next level of the software
- It preserves the benefit of a blackbox approach, in that it can generate these inputs rather quickly.

Alas, it still has disadvantages:

- The properties it can test for are still relatively shallow compared to what one could test by hand
- It requires some models of what constitutes a valid input

As a simple thought exercise, try the previous exercise of calculating the probability of an input reaching a particular bug, but now using a generation-based fuzzing approach. How much better does it do? A few percentage points? An order of magnitude? Many orders of magnitude?

BooFuzz is a generation-based fuzzer that allows you to encode the structure of input data, and then efficiently generate test inputs. It also helps with recording test cases and detecting crashes, instrumenting the target code, restarting the test target, etc. It's written in Python, so you can run it on Windows, Linux, etc.

Property-based fuzzing

In property-based fuzzing, the user provides a specification of the code in the form of *properties* that functions in that code ought to satisfy, and then the fuzzer tests that the properties hold in a large number of randomly generated cases. Specifications can be expressed in whatever language the fuzzer supports (e.g., QuickCheck uses Haskell, ScalaCheck uses Scala).

One way to think about property-based fuzzing is in relation to unit tests: Whereas unit tests allow you to test one instance of a property at a time, a property-based fuzzer works on your behalf to find multiple tests around that same property. Remember, the goal of testing is to find bugs, and the idea here is that you give the fuzzer the property that must hold, and the fuzzer tries to break it.

For example, in ScalaCheck, you give it a property about your code, and it attempts to disprove it. For example:

```
property("concatenate") = forAll { (a: String, b: String) =>
   (a+b).length > a.length && (a+b).length > b.length
}
```

will result in automatically finding that this property doesn't hold when the inputs are empty strings.

The greatest advantage of property-based fuzzing is that the tests are now a lot more meaningful: the fuzzer produces substantially better test cases that can penetrate arbitrarily deep into the code. How deep it goes depends on how well you write the properties.

The disadvantage is of course that, unlike dumb fuzzing, you need to think even deeper about the code now. Furthermore, the power of this fuzzer is limited by how well you write the property.

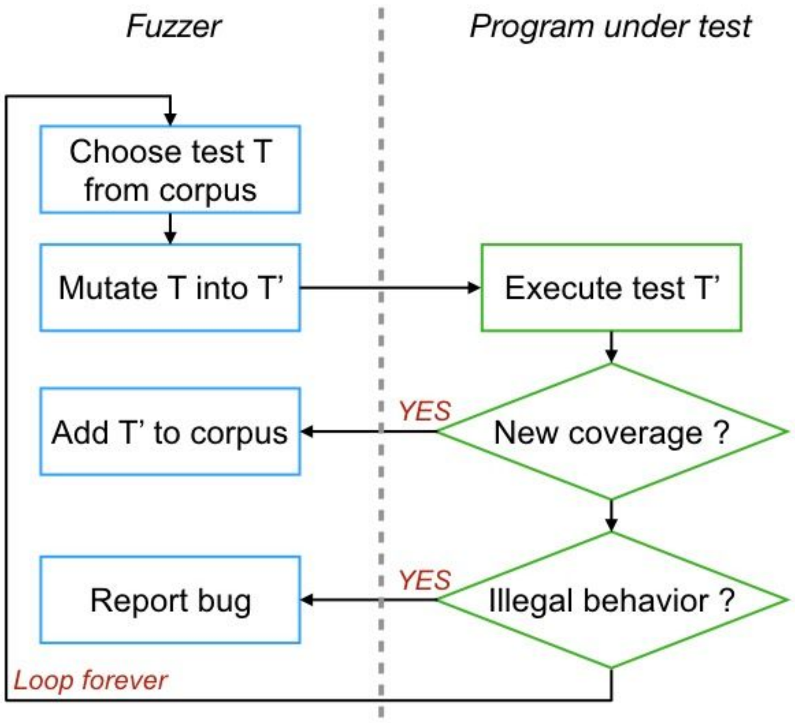
Mutation-based fuzzing

Mutation-based fuzzing consists of starting with well-formed inputs (often called "seeds") and repeatedly modifying them, more or less at random (depending on technique and tool), to produce new inputs. This preserves the benefit of blackbox testing while increasing the probability of inputs found in this way being “interesting” (e.g., capable of getting past the first layer of input parsing).

This is powerful enough to have found crashes and security vulnerabilities in software, with some of the most notorious security vulnerabilities having been found this way. See Microsoft's cloud fuzzing for Windows and Linux.

Detecting anomalous behaviors automatically during the test runs increases the chances of detecting the manifestation of a bug. Therefore, fuzzers check for a wide range of "illegal behaviors," with memory safety violations being the most popular. The premise is that higher-quality tests are more likely to find bugs.

Modern mutation-based fuzzers, like AFL or LibFuzzer, operate in a feedback loop like the one below:



The fuzzers rely on instrumentation in the target source code to detect program features triggered by tests, e.g., a basic block being executed or a buffer overflow. Whenever a feature is seen for the first time, the fuzzer reacts: it adds the test to its corpus of interesting test cases, or reports that it found a bug.

Different types of instrumentation can detect various features of interest. For example, *coverage bits* are on/off detectors of whether a particular edge in the control-flow graph (CFG) of the program is executed. The fuzzer instruments the target code to monitor each edge in the CFG, and when a coverage bit toggles from off to on, it tells the fuzzer that the last test input exercised new code. A *coverage counter* similarly detects how often an edge has been executed, and can signal to the fuzzer that it made progress when exploring a loop (in a loop, the same edge is traversed on each iteration of the loop, so the amount of increase in the coverage counter tells the fuzzer to what extent it is exercising further iterations of a loop). *Safety checks* detect abnormal conditions, alerting the fuzzer that a bug has been found. Such checks can be either added by developers in the form of assertions, or done automatically by tools such as Valgrind, UndefinedBehaviorSanitizer and ThreadSanitizer and AddressSanitizer in the Clang compiler toolchain. The GCC compiler has a `FORTIFY_SOURCE` option that does automatic bounds checking of dangerous functions to detect simple buffer overflows. `FORTIFY_SOURCE` does static and dynamic checks on buffer sizes for functions like `memcpy, strcpy, sprintf, gets`, etc. and if a fuzzed input triggers an overflow, it means that the fuzzer has found a potential security vulnerability. AFL’s libdislocator is a replacement memory allocator that aims to improve the odds of a fuzzer bumping into heap-related security bugs: it allocates buffers in special ways, it uses canaries, protects freed memory to catch use-after-free bugs, and so on.

Using such safety checks and then mutating test inputs to drive toward unexercised code increases the quality of tests as well as the speed with which the succession of tests drive toward finding a bug, and thus the number of bugs that fuzzers can detect. Without them, developers need to hope that illegal behavior leads to a segmentation fault or other visible exception. This does not always happen, particularly for tricky cases such as use-after-free or buffer over-read bugs.

Ideally, a fuzzer should simultaneously have high test throughput and high test quality, but unfortunately these two requirements conflict: obtaining good feedback comes at the cost of throughput. Both detecting program misbehavior and collecting code coverage information is done using program instrumentation, which competes for CPU cycles with the actual instructions of the program being tested. It is in fact not unusual for fuzzers to invest less than half of their resources into executing code of the target program, and spend the rest on improving test quality.

With the rise of machine learning (ML), researchers have employed ML techniques to do better mutation. Godefroid et al. used a neural-network-based statistical machine-learning technique to learn the input format from a set of valid inputs, then use this learned structure to generate new test inputs. This was applied to the PDF parser in Microsoft's Edge browser. Rajpal et al. use neural networks to learn from past fuzzing explorations not the structure but rather which byte to mutate in input files. Nichols et al. used GAN models help reinitialize the system with novel seed files. All this work was published cca. 2017.

Whitebox fuzzing

The idea of whitebox fuzzing is that the tool actually looks *at the code* it is testing and then uses that information to generate inputs. This can be a lot more precise. In fact, the most precise form of automatic code-driven test generation known today is dynamic test generation with symbolic execution. This is also the most advanced form of automated test generation, and it combines program analysis with testing, model checking, and automated theorem proving.

At a high level, a whitebox fuzzer will inspect the code and produce "custom" inputs that reach specific lines of code under specific conditions (e.g., with variables having a specific value). If for instance you see a line of code dereferencing a pointer, you may try to find an input that causes that pointer (or reference) to be null. Or if you see integer addition, you may want to find inputs that cause it to overflow. (Try this as a simple exercise on your favorite code).

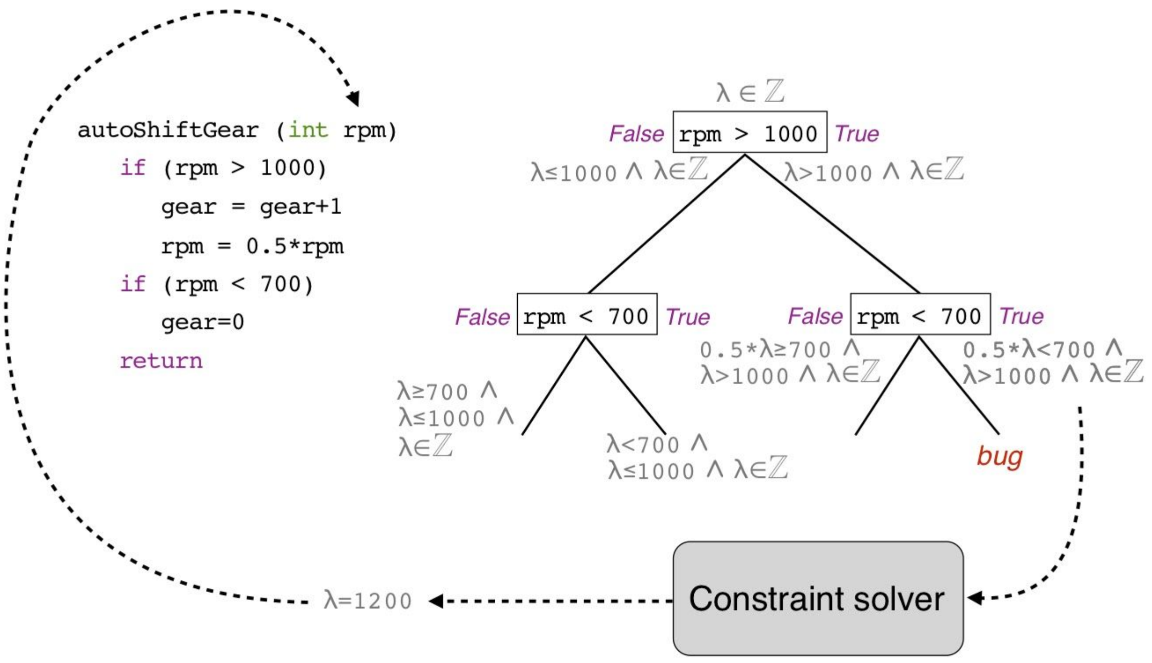
As we'll see below, whitebox fuzzing based on symbolic execution (SE) always generates "real" test cases, i.e., every bug report is accompanied by a concrete input that reproduces the problem (thus eliminating false reports). SE is powerful because it systematically checks each program path, and it does so exactly once, which makes it efficient (no work will be repeated as in other typical testing techniques). But of course (as we'll see) SE suffers from its fair share of scalability challenges. In fact, by now you must have already observed a trend: the smarter a fuzzing technique gets, the fewer tests it can run per unit of time. There ain't no such thing as a free lunch.

Before diving into the details of how this ultra-powerful technique works, let me note that there are some very compelling results to motivate us. For example, the tool called SAGE (used internally at Microsoft) is credited with having discovered on its own 1/3 of all Windows 7 security bugs, saving millions of dollars. SAGE is currently used daily for Windows, Office, and other Microsoft products. In the open-source domain, KLEE is an award-winning SE engine that has been used by hundreds of research projects leading to the finding of thousands of bugs. SE also has uses that go beyond testing, such as automatically deriving security exploits based on the patches for those vulnerabilities, or reverse-engineering proprietary device drivers.

Symbolic execution (SE) was first introduced as a program analysis technique in the 70s, but hasn't become truly useful until recently. Symbolic execution means executing a program with symbolic rather than concrete values. Assignment statements are represented as functions of their (symbolic) arguments, while conditional statements are expressed as constraints on symbolic values.

Symbolic execution (SE) can be used to symbolically explore the tree of all computations the program exhibits when all possible value assignments to input parameters are considered. In other words, SE explores all feasible executions, i.e., all program execution paths for which there exists an input that can exercise it.

As an example, consider the simple program on the left in the figure below, and its computation tree on the right.



```
autoShiftGear (int rpm)
   if (rpm > 1000)
      gear = gear+1
      rpm = 0.5*rpm
   if (rpm < 700)
      gear=0
   return
```

This program takes an integer value rpm as input. The set of possible values for program variable rpm is represented by a *symbolic* value λ that can initially take on any integer value: this is represented by the constraint λ ∈ Z. During symbolic execution of that program, whenever a branch depending on λ is encountered, a new constraint is generated to capture how to make that input-dependent branch condition evaluate to true (e.g., λ>1000) or false (e.g., λ≤1000) respectively. By repeating this process going down the tree, we obtain an execution tree annotated with conjunctions of input constraints which characterize what input values are required in order to reach what parts of the program. Those conjunctions of constraints are called *path constraints*, or *path conditions*, and are shown in grey.

During symbolic execution of a program, the whitebox fuzzer automatically checks for bugs. For example, one could instrument their code with UndefinedBehaviorSanitizer, ThreadSanitizer, AddressSanitizer, etc. as mentioned above to detect buggy conditions. When such a bug is encountered (shown in red above), the fuzzer takes the path condition (0.5*λ<700 ∧ λ>1000 ∧ λ∈Z in the above case) and feeds it to a constraint solver, which then finds a value for λ that satisfies this path constraint (e.g., λ=1200 in the above case). Here we are assuming that the condition of gear being set to 0 can be detected as a violation of some basic safety property.

What is this value λ=1200? It is an input value for the program that directly evidences the bug. The SE engine explored the code directly, zoomed in on the buggy line of code, and found a concrete input that makes the program execute that line of code in a way that violates the checked safety property. Voila, a compelling test case!

Said more formally, for each *control path* p, that is, a sequence of control locations of the program, a *path constraint* φ_p is constructed that characterizes the input assignments for which the program executes along p. All the paths can be enumerated by a search algorithm that explores all possible branches at conditional statements.

The paths p for which φ_p is satisfiable are *feasible* and are the only ones that can be executed by the actual program. The solutions to φ_p characterize the inputs that drive the program through p. This characterization is *exact* provided symbolic execution has *perfect precision*. This analysis amounts to a kind of exhaustive symbolic testing of all feasible control paths of a program[1].

What if, in the example above, `rpm` was not multiplied by 0.5 on the 4th line? In that case, the execution path that has True for the first if and True for the second if is not feasible, i.e., there does not exist an input for which that path can be followed. SE will not attempt to go down that path[2].

In practice, the key strength of symbolic execution is that it can generate quality test inputs that exercise program paths with much better precision than random testing or other blackbox heuristic-based test-generation techniques. Even though automatic test generation using symbolic execution suffers from several important limitations, approximate solutions are sufficient in practice. To be useful, symbolic execution does not need to be perfect, it must simply be “good enough” to drive the program under test through program branches, statements and paths that would be difficult to exercise with simpler techniques like random testing. Even if a systematic search cannot typically explore all the feasible paths of large programs in a reasonable amount of time, it usually does achieve better coverage than pure random testing and, hence, can find new program bugs.

So clearly SE can rapidly find the bugs and produce as test cases *only* those inputs that are really interesting. But SE also has drawbacks.

Summary

Let us recap what this lecture taught you:

- To test, you must get clear requirements from users
- Tests and requirements can be at any level: functions, modules, user interfaces…
- You can test at any stage of development
- You can evaluate your tests with coverage and performance
- Testing is not enough to guarantee that the software meets users’ needs
- Tests can be generated automatically with techniques like fuzzing and symbolic execution

[1] This is assuming that the theorem prover used to check the satisfiability of all formulas φ_p is sound and complete. For now, you can assume that it is, but in reality it is not.
[2] A powerful compiler would normally be able to figure out, for this particular program, that the *then* branch of the second *if* statement is dead code, and would eliminate it, and an SE engine may not even see it. But there are many situations in which this decision cannot be made statically (e.g., because it depends on inputs), in which case the compiler would not be able to identify the dead code.

The material from here on is advanced. We will not test on the exam your knowledge of what follows. But if you want to truly understand how the most advanced technique for test generation works, read on!

Path Explosion

First and foremost, naively using SE to symbolically execute all feasible program paths does not scale to large programs, because the number of feasible paths in a program can be exponential in the program size, or even infinite if the program, such as a network server, has a single loop whose number of iterations may depend on some unbounded input, such as a stream of network packets. A program like the Firefox web browser has more than 500,000 if statements; if just one thousandth of them were to have both a then and an else branch that are feasible for some inputs, then Firefox could be expected to have on the order of 2500 paths, which still far exceeds the number of atoms in the observable Universe.

There exist multiple solutions to this path explosion problem, but their specifics are beyond the scope of this lecture. Interested students can look into loop unrolling, symbolic summaries, and state merging. These are rather sophisticated techniques that are either taught in MS-level or PhD-level courses, or constitute the topic of active research.

An interesting alternative to making the SE engine more efficient is to instead transform the code so that SE has an easier time with it. In other words, we can present the same SE engine with an equivalent variant of the target program that is easier to symbolically execute. If this can be done automatically, then we're in luck.

Whereas a usual compiler translates programs into code that executes as quickly as possible on a target CPU, taking into account CPU-specific properties, a tool like Overify instead compiles programs to have the simplest possible control flow, using techniques like jump threading, loop unswitching, transforming conditionally executed side-effect-free statements into speculative branch-free versions, splitting objects into smaller ones to reduce opportunities for pointer aliasing, and others. The net effect is that compiling a program with the `-Overify` option reduces the time of exhaustive symbolic execution by up to almost two orders of magnitude.

Efficient Constraint Solving

Another key challenge in SE is the constraint solver, which is used to solve the path constraints. The solver gets used to produce the inputs that take to a target bug, but also during the symbolic execution itself, to determine whether following a branch is feasible or not. It is important to scalability to not follow an edge in the control-flow graph that is infeasible given the current path constraint. But this also mean that the constraint solver gets called quite often.

Much of what goes on inside a constraint solver can still be referred to as "black magic": they use sophisticated algorithms but also complex heuristics (which sometimes work, sometimes don't) and sometimes even random trial-and-error. The solver can often take a long time to come back with an answer. And when the solver times out, you don't know whether the constraint is satisfiable or not, you just know that, had you waited longer, you might have gotten a response. The reason one usually sets a timeout is because solvers often provide no guarantee of termination, especially for a formula that is unsatisfiable.

Fortunately, the science and engineering of automated theorem proving has made a lot of progress over the last decade as well. Notably, the past two decades witnessed the birth and rise of so-called Satisfiability-Modulo-Theories (SMT) solvers, which can efficiently check satisfiability of complex constraints expressed in rich domains. Such solvers have also become computationally affordable in recent years thanks to the increasing computational power available on modern computers.

Some solver challenges, however, cannot be overcome in a foolproof manner. Some program statements can result in path constraints that are just too hard to reason about symbolically. Consider for instance the following code:

```
int myFunction( String x ) {
   if ( SHA(x) == 4249821 )
      ... bug ...
   else
      return SUCCESS;
}
```

When symbolically executing this code, the SE engine would descend into symbolically executing the implementation of the SHA hashing function, which will result in extremely complex path constraints that need to be satisfied to reach the bug. In fact, the hash function makes it (by construction) very hard to find a value of x for which the hash equals some given value, so naturally the constraints are hard to solve. One could in principle use rainbow tables to come up with one, but solvers are not customized for such operations. So a standard SE engine executing this code will simply get stuck.

Interacting with the environment

In theory, symbolic execution does not use abstraction and is therefore fully precise: it generates “per-path verification conditions” whose satisfiability implies the reachability of a particular statement, so it generally does not produce false positives.

In practice, to test real-world programs, a symbolic execution engine must mediate between the program and its runtime environment, i.e., external libraries, the operating system, the thread and process scheduler, I/O interrupt events, etc. Normally this would mean that, for example, when a program opens a file, then the SE engine would have to symbolically execute the open() call, which in turn invokes a system call, which also needs to symbolically executed, and it may call into a device driver, which needs to be symbolically executed, and so on. Clearly this is impractical, and so symbolic execution engines need to minimize the time they spend executing the environment, while of course ensuring correct behavior of the program.

Existing solutions roughly fall into two categories: they either *concretize* the parameters of a call to the environment (i.e., pick a specific value for λ and execute the call with that specific value) and thus avoid symbolic execution of the environment altogether, or they abstract the environment as much as possible using models with varying degrees of completeness. We now discuss those two options.

Concrete Environment

The first modern symbolic execution engines (DART, CUTE, EXE) executed the program concretely, and maintained the symbolic execution state only during the execution of the program code itself. Whenever symbolic execution is not possible, such as inside external library calls (possibly executed in kernel-mode or on some other machine or process) or when facing program instructions with unknown symbolic semantics, the program execution can still proceed. This approach turns the conventional stance on the role of symbolic execution upside-down: symbolic execution is now an adjunct to concrete execution. As a result, a specific concrete execution can be leveraged as an automatic fall back for symbolic execution. This avoids altogether symbolic execution of the environment.

A benefit of this approach is that it can be implemented incrementally: only some program statements can be instrumented and interpreted symbolically, while others can simply be executed concretely natively. A tool developer can improve the precision of symbolic execution over time, by adding new instruction handlers in a modular manner.

The main drawback is that program behaviors that correspond to environment behaviors other than the ones seen in the concrete executions are not explored.

Modeled Environment

This drawback can be addressed with another approach: model the environment during symbolic execution. For example, KLEE redirects calls to the environment to small functions that understand the semantics of the desired action well enough to generate reasonable responses. With about 2,500 lines of code, they modeled roughly 40 Linux system calls, such as open, read, and stat. These models are hand-written abstractions of actual implementations of the system calls. Subsequently, Cloud9 expanded the models for KLEE to a full POSIX environment.

The benefit of this approach is that target programs can now be exposed to more varied behaviors of the environment, and their reaction can be evaluated. For instance, what does the program do whenever a write operation to a file fails due to a full disk? A suitably written model for write can have a symbolic return value, which will vary depending on the success or failure of that operation. In fact, one can write models that return different error codes for different failures, and a symbolic execution engine can automatically test the program under all these scenarios.

This approach has two main drawbacks. First, by definition, the model is an abstraction of the real code, and it may or may not model all possible behaviors of that code. (If the model was fully precise, it would be equivalent to the actual implementation.) Second, writing models by hand is labor-intensive and prone to error.

In-Vivo Symbolic Execution

To mitigate these drawbacks, selective symbolic execution does not employ models but instead automatically abstracts the environment. In doing so, it is guided by consistency models that govern when to over-approximate and when to under-approximate.

S2E is a symbolic execution engine for x86 and ARM binaries that was developed in DSLAB, built around a modified version of the QEMU virtual machine. It dynamically dispatches guest machine instructions either to the host CPU for native execution or to a symbolic execution engine embedded inside S2E. Most instructions do not access symbolic state, so they can run natively, while the rest are interpreted symbolically. The original use case for S2E was testing full system stacks with complex environments (proprietary OS kernel, many interdependent libraries, etc.). An accurate assessment of program behavior requires taking into account every relevant detail of the environment, but making a priori the choice of what is relevant is hard. So S2E offers "in-vivo" symbolic execution, in which the environment is automatically abstracted on-the-fly, as needed. This is in contrast to "in-vitro" symbolic execution, as done by symbolic execution engines that model the environment.

S2E was used both in industrial and academic settings. Engineers at Intel used it to search for security vulnerabilities in UEFI BIOS implementations. The DDT tool used S2E to test closed-source proprietary Microsoft Windows device drivers, without access to the driver source code or inside knowledge of the Windows kernel (it found memory leaks, segmentation faults, race conditions, and memory corruption bugs in drivers that had been shipping with Windows for many years). Last but not least, two of the seven systems competing in the finals of DARPA’s Cyber Grand Challenge in 2016 were based on S2E. This competition was an all-machine computer security tournament, where each competing machine had to autonomously analyze computer programs, find security vulnerabilities, fix them, and launch attacks on other competitors. As part of Galactica (one of the DARPA competitors), S2E launched 392 successful attacks during the competition, twice as many as the competition’s all-around winner.

Automated Testing in the Cloud

An orthogonal approach to speed up symbolic execution is parallelization of path ex- ploration on a cluster of machines, harnessing its aggregate CPU and memory capabilities (alternatively, one could imagine running a symbolic execution engine on a super- computer). One way to parallelize symbolic execution is by statically dividing up the task among nodes and have them run independently. However, when running on large programs, this approach leads to high workload imbalance among nodes, making the entire cluster proceed at the pace of the slowest node—if this node gets stuck, for in- stance, while symbolically executing a loop, the testing process may never terminate. In Cloud9, a method is used for parallelizing symbolic execution on shared-nothing clusters in a way that scales well. Without changing the exponential nature of the problem, parallel symbolic execution harnesses cluster resources to make it feasible to run automated testing on larger systems than would otherwise be possible.

In essence, software testing reduces to exercising as many paths through a program as possible and checking that certain properties hold along those paths (no crashes, no buffer overflows, etc.). The advances in symbolic execution lead naturally to the "testing as a service" (TaaS) vision of (1) offering software testing as a competitive, easily accessible online service, and (2) doing fully automated testing in the cloud, to harness vast, elastic resources toward making automated testing practical for real software. A software-testing service allows users and developers to upload the software of interest, instruct the service what type of testing to perform, click a button, and then obtain a report with the results. For professional uses, TaaS can integrate directly with the development process and test the code as it is written. TaaS can also serve as a publicly available certification service that enables comparing the reliability and safety of software products.