Sigrid Eldh and Serge Demeyer

The Next level of Software Test Automation

Preprint of a bookchapter, to be published by Springer

March 9, 2025

Springer Nature

Contents

Chapter 1
Exploiting Test Automation

Abstract Modern software development deploys software into production using a fully automated build pipeline. In this chapter we explain how to exploit the such a build pipeline to evaluate and improve the tests. Once the system under test is adequately tested, one can uncover under tested areas by manipulating either the system under test or the test suite. And if the tests are automated, it is possible to do so so at an unprecedented scale. As such we cover techniques such as mutation testing, fuzz testing, test amplification, resilience testing, automated program repair, automated GUI testing, metamorphic testing, A/B testing.

Learning Outcomes

After reading this chapter, you should . . .

1. Understand the role and stages of an automated build-pipeline.
2. Explain a series of advanced test automation techniques.
3. Incorporate advanced test automation techniques into an automated build pipeline.
4. Assess the costs and benefits of introducing advanced test automation techniques.

Upon completing the exercises at the end, you will be able to . . .

1. Calculate the mutation coverage of a given test suite (mutation testing).
2. Expose potential crashes in a system under test (fuzz testing).

1.1 Build Pipelines

In current software engineering, the automated tests are combined into a fully automated build pipeline. The key for such a build pipeline (as illustrated in Figure 1.1) is a tight feedback loop where a series of increasingly powerful automated checks scrutinize every code change, to ultimately deploy the approved changes into the actual system.
All these automated checks serve as quality gates, safeguarding against software faults.

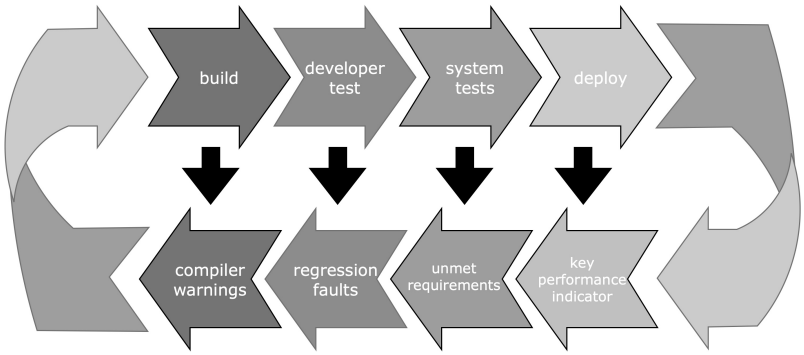

Fig. 1.1 Automated Build Pipeleins

- The (incremental) build of the system issues alerts with varying levels of severity. These range from compiler issues (where the system under test will not even build) to static program analysis warnings (where unsafe code constructs are flagged).
- The developer tests (unit and component tests) find regression faults demonstrating that the code change broke something which worked previously.
- The system tests (verifying the functional and non-functional requirements) demonstrate whether the system deviates from its specifications.
- The system gets deployed on the actual system. Telemetry collects measurements on how the system behaves in production and compares them against the key-performance indicators established for the system under test.

In this chapter we explain how to exploit the facilities inherent in such a build pipeline to evaluate and improve the quality of the flow itself. Indeed, once the system under test is adequately tested, one can uncover under tested areas by manipulating either the system under test or the test suite. And if the tests are automated, it is possible to do so so at an unprecedented scale.

In the next sections we describe techniques to improve the system under test (i.e., reveal yet unknown faults) or to strengthen the test suite (i.e., expose under tested aspects of the system).

- **Mutation testing** comes early in the build pipeline, during developer tests (i.e. unit and component tests). It measures the strength of the tests suite by injecting artificial faults (called mutants) into the code under test and counting how many of them are caught by the test suite.
- **Fuzz testing** comes later in the build pipeline, at the stage of automated system tests. Fuzzing identifies security vulnerabilities by exercising the system under test with random input (the fuzzing) to force unresponsive states, such as crashes or freezes.
- **Resilience Testing** (a.k.a. Fault injection, Robustness Testing) comes at the very end when the system is in the pre-release phase. Here we verify the robustness of the system under test via a “what-if” analysis. We inject faults in critical com-

ponents or connectors and verify whether the system still provides an acceptable service level.

We also briefly touch upon a few other techniques which are on the radar.

- **Test amplification** takes an automated test suite (developer tests or system tests) and manipulates the test code (via amplification operators) to increase the coverage.
- **Automatic Program Repair** automatically generate patches on a system under test in order to make a failing test suite pass.
- **Automated user-interface testing** (a.k.a. Monkey testing) expose the user-interface of the system under test with random input to see whether it behaves as expected.
- **Metamorphic testing** addresses the oracle problem by defining properties (metamorphic relations) on the expected output and verifying whether the actual output satisfies these metamorphic relations
- **A/B Testing** compares the performance of two versions of a system to see which one appeals more to users.

Readers should be aware that all the above draws on principles and inspiration drawn from subfields within the testing community. We explicitly list them below.

- **Search based testing** which uses AI based search algorithms to tackle the "needle in a haystack" problem permeating the whole testing life cycle.
- **Model-based test generation** where of model of the system under test is used to exercise lurking faults.
- **Random testing** where the system under test gets exposed to random input in the hope of triggering unexpected conditions.

1.2 Mutation Testing

1.2.1 What?

Mutation testing (also called mutation analysis — within this book the terms are used interchangeably) is the state-of-the-art technique for evaluating the fault-detection capability of a test suite [9]. While the principle is applicable to any automated test suite, it is mainly used for unit and component tests. The technique deliberately injects faults into the code and counts how many of them are caught by the test suite.

As with every field of active research, there is a lot of terminology. Below we list the most important terms readers should be familiar with.

Mutation Operators. The faults injected into the code are based on mutation operators. A mutation operator is a source code transformation which introduces a change into its input. Typical examples are replacing a conditional operator ($>=$ into $<$) or an arithmetic operator ($+$ into $-$).

Killed and Survived (Live) Mutants. Mutation testing first creates a defective version of the software by artificially injecting a fault based on one of the known mutation operators (*Mutation*). After generating the defective version of the software (*Mutant*), it is passed onto the test suite. If a test fails, the mutant is marked as killed (*Killed Mutant*). If all tests pass, the mutant is marked as survived or live (*Survived Mutant*).

Mutation Coverage (*Mutation Score*). The whole mutation analysis ultimately results in a score known as the *mutation coverage*: the number of mutants killed divided by the total number of mutants injected. A test suite is said to achieve *full mutation test adequacy* whenever it can kill all the non-equivalent mutants, thus reaching a mutation coverage of 100% Such test suite is called a *mutation-adequate* test suite.

Special Cases. Unfortunately, the distinction between killed and survived mutants it too simplistic for practical use. The community also distinguishes the following special cases.

- *Invalid Mutants.* Mutation operators introduce syntactic changes, hence may cause compilation errors in the process. A typical example is the arithmetic mutation operator which changes a '+' into a '−'. This works for numbers but does not make sense when applied to the C++string concatenation operator also denoted with a '+'. If the compiler cannot compile the mutant for any reason, the mutant is considered invalid and is not incorporated into the mutation score. In programming languages with dynamic typing (such as Python, JavaScript) invalid mutants are hard to discern.
- *Equivalent Mutants.* Some mutants do not change the semantics of the program, i.e. its output is the same as the original program for any possible input. Therefore, no test case can differentiate between a so-called equivalent mutant and the original program. Equivalent mutants should be excluded from the mutation coverage. Unfortunately, the detection of equivalent mutants is undecidable due to the halting problem. Therefore, it is left to the software engineer to manually weed out equivalent mutants. For a full-scale mutation analysis such a manual check is seldom done, and equivalent mutants are then considered a threat to validity.
- *Redundant ('Subsuming') Mutants.* In principle, one test case will kill several mutants. However, some mutants are easy to kill while others are not. Take for instance test case A killing mutant X (easy to kill) and mutant Y (difficult to kill) and test case B which only kills mutant X. Then mutant Y is said to subsume X if any test that kills Y will also kill X. This implies that mutant X is redundant. Redundant mutants are undesirable, since they waste resources and add no value to the process. In addition, they inflate the mutation score because often it is easy to kill many redundant mutants just by adding a single test case.
- *Infinite loops.* Some mutants induce an infinite loop into the program under test. Therefore, most mutation tools abort the program under test when it runs an order of magnitude longer than expected and mark the corresponding mutant as 'killed'. Note that this assumption is not always correct, as in rare occasions the

mutant can take much longer to be analyzed due to other circumstances such as IO problems, ... In such cases, the mutant should be counted as 'survived', but automatic detection of these scenarios is undecidable due to the halting problem.

- *Flaky Tests.* Mutation testing assumes tests to be completely deterministic: every test run should produce the exact same output. The presence of flaky tests — whose outcome can non-deterministically differ even when run on the same code under test— invalidates this assumption. When a test suite contains flaky tests, the mutation analysis is unpredictable, as some mutants might be killed when in fact the failing tests are due to flakyness and not the injected fault itself; which in turn is a threat to validity. Sometimes a mutant will also introduce flakiness and that complicates matters even further.

1.2.2 Why? (Statement Coverage versus Mutation Coverage)

Given the widespread reliance on automated tests, lots of research went into measuring the quality of tests. Usually test quality is measured in terms of code coverage, i.e., the proportion of code that is executed by the test suite. The measures most often used are statement coverage and branch coverage. Statement coverage is the number of statements in the program that are executed at least once by the test suite divided by the total number of statements. Similarly, branch or sometimes also called decision coverage is the number of branches (decisions) in the control flow executed at least once by the test suite divided by the total number of branches. Branch coverage subsumes statement coverage, because if all branches are examined, all statements contained in the branches are examined in the process. For safety critical code MC/DC is often required, which is not only branch coverage —but in combination with what is called 'basic condition'— meaning that every value should be tested for its 'impact' on the branch outcome.

False confidence. Unfortunately, achieving a high code coverage may give a false sense of confidence on the strength of the test suite if using such a weak coverage as statement coverage. Because it is not enough for the test to execute the code under test, it should also verify whether the result corresponds to the expected value. An *assertionless test* is a common test smell, referring to a unit tests which does not verify whether the results are as expected. Such assertionless test are an easy way to boost the test coverage when under time pressure. One writes a test which illustrates a 'happy day' scenario of how the unit under test is supposed to be used, yet never verify the intermediate state. The code coverage will easily reach 80%, however the assertionless test will only fail when an unhandled exception occurs. Because it is easy to e.g., fulfill statement coverage with 'any' data —but maybe not a defect revealing data point (e.g., on the boundary) it is very important to first attempt to write good smart tests (choosing data carefully)— and then use coverage as an 'assessment' of how good you were at writing your tests, and learn how come you did miss those last 20% Otherwise it is easy to loose the value it can bring.

Listing 1.1 Code sample for a C++function with a fault in line 02

```
01      int findLast(std::vector<int> x, int y) {
02          for (int i = x.size() - 1; i > 0; i--)
03              if (x[i] == y)
04                  return i;
05          return -1;
06      }
```

Listing 1.2 Two tests for findLast; 100% coverage yet fault is not exposed

```
07      TEST(FindLastTests, ReachNotInfect) {
08          EXPECT_EQ(3, findLast({1, 2, 42, 42, 63}, 42));
09      }
10
11      TEST(FindLastTests, ReachInfect) {
12          EXPECT_EQ(-1, findLast({1, 2, 42, 42, 63}, 99));
13      }
```

Reach — Infect — Propagate — Reveal (RIPR) criterion. Code coverage metrics only tell us which parts of the system have been Reached by the tests (with any data). They do not assess whether the program state is Infected, nor whether the fault gets Propagated as an observable difference and eventually Revealed by an assert statement. This Reach — Infect — Propagate — Reveal criterion for an effective test case is called the RIPR model, providing a fine-grained framework to assess weaknesses in a test suite [10].

RIPR: an example. Consider the code sample in Listing 1.1. It is a C++function which scans a vector from the last position to the first position, looking for an element y. It returns the position of the element if it is found and otherwise returns -1. However, there is a subtle fault in line 02: the '$i > 0$' should read as '$i >= 0$'.

Now consider the test cases (written in the gtest framework) in Listing 1.2. The first one tests for the normal case, an element which is in the vector and asserts (via the `EXPECT_EQ` call in line 8) that the position of element 42 (which appears twice in the vector) should be 3. The second one tests for the case where an element is not present and asserts that the result should be -1. Together these two tests have a 100% statement coverage and even branch coverage. Nevertheless, they do not expose the fault.

We need to add one additional test —shown in Listing 1.3— to address this weakness in the test suite. The first test (lines 14 — 16) propagate the fault, however we need the assert in line 18 to actually reveal the fault. We expect that element 1 will be found in position 1; but it will return -1 hence the test fails.

Listing 1.3 Two extra tests for findLast. Only the last actually reveals the fault.

```
14      TEST(FindLastTests, ReachInfectPropagate) {
15          findLast({1, 2, 42, 42, 63}, 1);
16      }
17      TEST(FindLastTests, ReachInfectPropagateReveal) {
18          EXPECT_EQ(1, findLast({1, 2, 42, 42, 63}, 1));
19      }
```

```
[ RUN ] FindLastTests.ReachInfectPropagateReveal
tests.cpp:23: Failure
Expected equality of these values:
  1
  findLast({1, 2, 42, 42, 63}, 1)
    Which is: -1
[ FAILED ] FindLastTests.ReachInfectPropagateReveal
```

1.2.3 How? (Fewer, Smarter, Faster)

Prerequisites. Assuming you want to adopt mutation testing in your automated test process, you must know that there are a few prerequisites to be fulfilled.

- *Automatic.* You need a test suite which executes fully automatically. This excludes exploratory testing, user acceptance testing and all other tests which involve a human in the loop to interpret the results of the test.
- *Deterministic pass/fail verdict.* Executing the test suite should result in a clear, simple and deterministic verdict: pass or fail. This excludes typical performance tests, penetration tests, load tests, ... Flakey tests (tests which sometime pass or sometime fail when run on the same code under test) should be avoided at all costs.
- *Fast.* The tests execution should be fast. We are speaking of seconds are a few minutes at best. Remember the test suite will be executed for each and every mutant. This typically excludes hardware-in-the-loop-tests and user-interface tests.
- *Flexible build system.* The build system that you use should be fairly flexible to integrate the mutation analysis. In particular, mutation analysis considers a broken build a good thing (the mutant is killed) while in normal circumstances broken builds cannot be tolerated.

Steps. Mutation testing —in its basic form— requires a tremendous amount of computing power: for each injected mutant, the code base must be compiled and tested separately. Below is the list of steps involved in a mutation analysis without any optimisations.

- Step 0: *pre-phase*. The software system is built without compilation errors and all software tests succeed.

- Step1: the *mutant generation* phase enumerates all source files and applies the mutation operators on all applicable locations.
- Step 2: the *mutant compilation* phase creates an executable test for one mutated version of the system. This may result in an invalid mutant.
- Step 3: the *mutant execution* phase runs the test suite and records whether all tests pass or at least one test fails. Infinite loops must be aborted.
- Step 4: at the end, a *report* is generated listing which mutants were (not) killed and the mutation coverage for the test suite.

Do fewer, do smarter, and do faster. A lot of research is devoted to optimizing the mutation testing process, summarized under the vision: do fewer, do smarter, and do faster [12]. One can for instance inject mutants in the byte code or compiled code to remove the costly compilation step. One can prioritize the tests so that the ones with the highest likelihood of failure are executed first. One can select a semi-random subset of mutants, hoping that the results of the subset will be a good approximation for the whole. There are two techniques however that have the highest impact: parallel mutation testing and incremental mutation testing.

Parallel (cloud-based). The whole mutation analysis is inherently parallel. Once a mutant is generated all subsequent steps can run independently from one another. Early on, several attempts have been made to run mutation testing on parallel hardware. The current trend is to run the mutation analysis in a distributed network set-up where one can scale the parallel mutation analysis.

Incremental (pull-request based). The most effective way to scale the mutation analysis is to inject fewer mutants. A natural way to do so is an incremental approach by limiting the scope of mutant generation to sections of code which have changed since the last mutation run. This reduces the number of mutants significantly yet is bound to omit a few relevant mutants. Given the modern workflow centered around git, it is only natural to scope the sections of code where to inject mutants in the code included in the pull request.

1.2.4 Where? (Industrial Adoption)

Within academic circles, mutation testing is a popular research topic. However, it took a while for the software industry to adopt. At the time of writing there is sufficient evidence that mutation testing —despite its computational cost– indeed helps to find weaknesses in a test suite. Below we list a few of the reported success stories, including the ones we experienced ourselves within the TESTOMAT project.

Safety Critical — Airborne. In 2013, Baker and Habli reported on an application of mutation testing for safety critical software (airborne systems) using high-integrity subsets of C and Ada [3]. They noticed that mutation testing is a nice complement to

the MC/DC coverage analysis required for the highest safety integrity level. To speed up the process, they applied parallel mutation testing across a test farm of desktop PCs. They discovered that many of the surviving mutants were equivalent mutants (i.e. impossible to kill) representing false positives of the mutation analysis.

» Despite this poor precision the authors concluded that these helped to indicated weaknesses in the test case design, *"with poor test data selection and overcomplicated tests being the main contributors to inadequate mutation scores. [. . .] mutation testing demonstrates exactly where the testcase set can be improved."* [3].

Safety Critical — Mechatronic. In 2017, Ramler et. Al reported on a second application of mutation testing on an industrial scale, this time for a mechatronic system written in C and comprising 60,000 lines of code [15]. Unit testing was deemed important and 100% MC/DC coverage was set as a goal. From a practical standpoint, the authors confirmed that it is a non-trivial task to integrate the mutation analysis into the development pipeline. To handle the scale of the mutation analysis (a total of 4071 hours), they parallelized the test execution cycle and distributed it across a cluster of standard desktop PCs. The whole mutation analysis resulted in 27,158 live mutants. They manually sampled 200 of them and 24% of them were classified as equivalent mutants (false positives). The time to manually review each mutant was recorded as well and was quite reasonable: 2 minutes on average and 20 minutes at most.

» The authors conclude that a mutation analysis provides actionable suggestions for the test engineers. *". . . mutation testing provides hints about deficiencies in test cases that are otherwise hard to discover. The feedback can be directly used for revising and enhancing the tests."* [15].

Pull Requests — Google and Facebook. Tricorder — the code review infrastructure within Google— illustrates the state-of-the-practice for modern code review [16]. Facebook has a similar "diff-based" code quality analysis infrastructure [4]. Upon the creation of a pull-request, a series of quality checks are applied on the code fragments within the pull request and the results are subsequently passed to two expert code reviewers. The quality checks include a mutation analysis, which is only applied on those code fragments included in the pull request [14].

» The authors argue that mutation adequacy (i.e. 100% mutation coverage) is too expensive, but that the mutation analysis provides actionable hints for the programmer to strengthen the test suite. *". . . we have observed many cases in which mutants caught actual bugs in the code, and even more where test suites were improved to kill the reported mutants."* [14].

Refactoring opportunities — NFluent. In 2019, Cyrille Dupuydauby (a software engineer at NFluent) wrote an experience report about integrating Stryker (a mutation

tool for .Net programs) in their development pipeline [6]. The team had test coverage tools in place and aimed for 100% line and branch coverage. The report confirmed that mutation analysis is slow (4 hours in their case) yet the first analysis revealed a mutation score of 224 / 1557 = 14,3%. Most interestingly, almost all live mutants (even the equivalent ones) were relevant and were strong indications of actual code weaknesses. The author gives several concrete examples where the unit-test itself was strengthened because of the mutation analysis. But equally important he provides several examples where the code in the system under test was too complex; refactoring allowed for easier test cases which subsequently increased the mutation score.

» The author concludes that *"(1) At the very least it will show you how much risk remains in your code and help you identify where you should add some tests. (2) If you have decent coverage, it will help you improve your code and your design."* [6]

1.3 Fuzz Testing

1.3.1 What?

Fuzzing (or Fuzz Testing) is a technique to identify vulnerabilities in a system-under-test [11]. Valid input is replaced by random values with the goal to force the system-under-test into unresponsive states, such as crashes or freezes. From a testing perspective, crashing the system is considered a serious issue as the system should be able to handle an unexpected situation gracefully instead of terminating abruptly. Moreover, crashing issues could lead to vulnerabilities for malicious users to exploit. For example, an attacker could exploit a crashing issue to cause a denial of service on a web service. Therefore, Fuzz Testing is an important technique in the arsenal of software engineers.

A common way to classify fuzzing tools is the kind of information they exploit to force the program under test into unresponsive states.

- **Input Grammar.** Classic fuzzing approaches only rely on the structure of the input data. The general idea is to give random or invalid input data to the program and watch how it responds. However, most random inputs get rejected in the early layers of the program hence are considered a waste of resources. For a more efficient analysis, fuzzers use the input grammar of the language and apply mutation operators to create malformed data at the boundaries of what is acceptable or not. (Cfr. Equivalence Partitioning and Boundary Value Analysis).
  This category is often referred to as *black-box fuzzing*, because the fuzzer explicitly neglects information about the system under test to maximize the chance of revealing omissions in the requirements.
- **Program Analysis.** This type of fuzzing in contrast does have access to the source code and relies on program analysis and the knowledge of the program's internal

structure to generate semi-random input data that exercises various control or dataflow paths. Two flavors of program analysis are (i) symbolic execution and (ii) abstract interpretation, techniques which divide the search space of possible execution paths in equivalence partitions. (Cfr. Equivalence Partitioning).
This category is often referred to as *white-box fuzzing*, because the structure of the program under test is used to maximize the code coverage.

- **Instrumentation.** Some fuzzers use light instrumentation techniques to create logs of the execution path for the corresponding input. They exploit this knowledge to produce inputs that force the program under test in varying execution paths, as such making observations on which branches causes the system to crash. Coverage-guided fuzzing is an example of a fuzzing techniques, which looks for increasing program coverage to force crashes.
  This category is often referred to as *grey-box fuzzing*, because it situates itself in between white-box and black box fuzzing.
- **Hybrid.** To compensate for the advantages and disadvantages of the aforementioned approaches, recent tooling adopts a hybrid strategy, combining the knowledge extracted from the input grammar with the data and control flow graphs extracted from the program analysis or instrumentation. A well-researched subfield is termed *concolic fuzzing*, named after the combination of CONC-rete and symb-OLIC execution. In this case, code instrumentation is used to obtain execution traces while symbolic execution is used to derive related constraints.
- **Ensemble fuzzing.** Another way to offset the strengths and weaknesses of a given fuzzing approach is to have different fuzzers collaborate on the same fuzz target. The presence of build pipelines greatly facilitates such ensemble fuzzing.

A more actionable way of classifying fuzzing tools is the attack surface which is systematically exposed with malformed input. This is the common classification used to select the fuzzing tool to be incorporated in your build infrastructure.

- **Rest API.** Web-services need to provide an interface to their clients to access their resources and a REST API is one of the dominant techniques for doing so. REST provide standard database functions like creating, reading, updating, and deleting information (also known as CRUD) concerning the resource they provide access to. Obviously, such CRUD exposes to the system to security attacks. Fuzzers then systematically expose the REST API to unexpected combinations of CRUD operations. A good example is the *RESTLER fuzzer* [2].
- **Protocols.** Distributed computer systems communicate over a network by sending and receiving messages. These messages adhere to certain conformance criteria and follow a certain protocol that synchronizes state transitions in the respective systems. Security attacks alter the contents of these messages, as such jeopardizing the correct synchronization. Protocol fuzzers reveal where the a distributed systems is vulnerable to malicious changes in the protocol. *TCP-Fuzz* is an example fuzzing tool for the TCP protocol [19].
- **File Format.** Many applications receive input via files. File format fuzzing provides the fuzzer with a legitimate file, and then repeatedly mutates the contents

of the file to feed it into the application under test to see whether it behaves unexpectedly (crash, freeze, memory leaks). *QuickFuzz* generates unexpected inputs of common file formats for third-party software [8].
- **Kernel.** Operating systems must function in a wide variety of situations, hence robustness and security are primary concerns. Kernel fuzzers exercise the API of the operating system with random sequences of valid (and invalid) calls to expose vulnerabilities. *syzkaller* is the de facto tool for fuzzing the Linux kernel [5].

Related Techniques. Fuzz testing has a lot in common with *random testing*; some people even argue that random testing is a subset of (white box) fuzz testing. With random testing, inputs are selected from the entire space of possible inputs; no attempts are made to steer the random choice. The argument is that any attempt to steer the choice rests on an assumption that might be false for the software under test. Therefore, the extra analysis time invested may not necessarily reveal critical bugs.

1.3.2 Why? (Security Testing)

Security faults are a subset of all faults, but they have grown in importance since almost all software systems are somehow connected to the internet, hence may be exposed to attacks. Information security, according to ISO 27001 standard, consists of the preservation of information confidentiality, integrity, and availability, as well as of the systems involved in its treatment, within an organization. It is beyond the scope of this report to discuss the numerous aspects of security testing as many of them —ethical hackers, penetration testing to name but a few— have a large manual component. However, to complete the technical operational details of security testing in the software development life cycle, automated tests have a lot to contribute. Fuzz testing is one of the prominent approaches therein. Indeed, once automated test execution is in place, it is possible to tirelessly enumerate different approaches. Fuzzers systematically expose the system under test to invalid, malformed, or unexpected inputs to reveal vulnerabilities. If successful, fuzzers will demonstrate that it is possible to access protected data, or that input channels may be blocked or flooded.

1.3.3 How?

Prerequisites. Assuming you want to adopt fuzzing in your automated test process, you must know that there are a few prerequisites to be fulfilled to successfully launch a fuzzing campaign.

- *Control input & output.* You need to have a strong level of control over the system under test. You should be able to launch the system, provide the necessary input and observe the system state (crash, freeze, infinite loop, corrupted memory, . . .).
- *Oracle.* You need a deterministic oracle to decide whether a given output (system state) represents a fault. This is far from trivial for memory related faults (read before write, use after free, . . .).
- *Fuzzing Harness or Fuzz Wrapper.* You need an adapter between the fuzzer and system under test. Applications that process data directly from a file or console input channel can most likely be fuzzed directly. For other cases —a typically lightweight— fuzzing harness is necessary to route input data from the fuzzer the interface of the system under test.
- *Configuration.* Depending on the kind of fuzzer you use, you need to provide the proper configuration to steer the fuzzer towards the most vulnerable targets and to decide on whether the output is acceptable or not.

Steps. Once automated test execution is in place, incorporating a fuzzer into the build pipeline is rather straightforward. Below we describe the most important steps to set-up a successful fuzzing campaign.

- Step 0: *pre-phase*. The software system is built without compilation errors and all software tests succeed.
- Step 1: Install the *fuzzing harness* to route the input data to the system under test.
- Step 2: *Configure* the fuzzer with the appropriate target and oracle. Choose the available *time budget*.
- Step 3: Execute the *fuzzing campaign* for the given time budget.
- Step 4: *Interpret the results*. Replicate the suspicious runs and identify the root cause. Fix the system under test to deal with the fuzzing input causing malicious behavior.

Pros and Cons. Once automated test execution is in place, incorporating a fuzzer into the build pipeline is rather straightforward. However, as always it comes with some advantages and disadvantages.

- *Replication.* One of the most prominent advantages is that when a fuzzer demonstrates that it can crash or freeze a system, one can easily replicate the steps to bring the system into the erroneous state. This allows for debugging the system under test to identify and repair the root cause of the fault.
- *Autonomous.* The fuzzer acts as a robot, independently searching for vulnerabilities, with no manual/human intervention. It can do so for as long as needed or for as long there is server capacity available.
- *Lack of explicit oracle.* Fuzzing only detects exceptional program states (crashes, freezes, etc.). There are no explicit assertions verifying whether the malformed input causes incorrect output. Hence, fuzzing will only detect a very small fraction of bugs.

- *Time budget.* When incorporating a fuzzer into the build pipeline it will essentially run forever, reporting all vulnerabilities it finds. In theory this is great, in practice one must set some upper limit to how long the fuzzer should execute.

1.3.4 Where?

Fuzzing is a very intuitive testing approach hence has a high appeal in practice. There is plenty of evidence that this technique works on an industrial scale. More recently there are several success stories where fuzzing is part of the automated build pipeline.

Standards. Fuzzing is a recommended practice in atseveral standards for safety-critical systems.

- ISO/SAE 21434—Road vehicles—Cybersecurity Serge ►*Verify standards – ISO/SAE 21434 – ISO 26262—Road vehicles—Functional Safety – ISA/IEC 62443-4-1 - Secure product development lifecycle requirement – IEEE reliability draft standard by Sigrid*◄

Trophies. Many of the open-source fuzzing tools explicitly list the defects that have been identified and fixed by means of their tool. To give an idea: as of August 2023, OSS-Fuzz has helped identify and fix over 10,000 vulnerabilities and 36,000 bugs across 1,000 projects [13]. Many testimonials on bug bounty programs that have been won by adopting various fuzzing tool. SQLlite for instance is a popular target for third-parties to fuzz and the team behind the tool report to occasionally receive (and fix) bug reports found by independent fuzzers [17].

Embedded systems. Eisele et. al. argue that embedded systems is a growing interest area for fuzz testing [7]. However, it is noted that within embedded systems compilers have various options for controlling the actual code deployed on the target device.

» The authors concludes that *"fuzzing [. . .] is not an effective means of testing code portions that interact directly with the hardware; incidentally, because of the diverging compiler and environment, this would not test the exact code that ends up on the actual device..'* [7]

Automotive. Werquin et. al. argue that modern cars contain a staggering amount of software. This comes at a risk: *"Since 2010, a series of high- profile attacks illustrate that with increased vehicular connectivity even remote adversaries can take control of critical functions of a vehicle."* [18]

» The authors argue that fuzz testing is a promising avenue, however comes with some challenges. *"A key difficulty to overcome here is the definition of oracle functions that define when a fuzzer has potentially triggered a bug or at least*

an 'interesting' system state, and to automatically evaluate these functions. [. . .] While experimenting with fuzzing strategies, we observed that an ECU's response to a message is often delayed. During the delay period, other messages are being sent by the fuzzer, which makes identifying the CAN messages specifically responsible for an response more difficult." [18]

1.4 Resilience Testing

1.4.1 What?

Fault injection testing verifies the robustness of a system-under-test. It deliberately injects faults into the system under test and observes the system's behavior [1]. In contrast with mutation testing this is not done to assess the strength of the test suite but to verify whether the system itself withstands stressful situations.

1.4.2 Why? (Fault Tolerance)

Modern software systems are built out of countless components, many of them originating in various sources and residing in different locations. If one of these components fails the whole system may be compromised, hence software engineers want their system to withstand such failures. If a fault occurs, the system should still provide an acceptable service level. Ideally, the system should gracefully recover when the failing component has resumed its operation.

1.4.3 How? (What - Where - When - How)

When injecting a fault, a test engineer must make four decisions.

- What to inject? (= the fault's type).
- Where to inject? (= the fault's location).
- When to inject? (= the activation time).
- How to assess the acceptable service level? (= the oracle).

To make these decisions, a thorough risk analysis of the system under test and its architecture is needed. Usually, this risk analysis is driven by a failure mode and effect analysis (FMEA), a review process (a) identifying all possible failures in a system design, (b) the consequences of such failures and (c) the mitigation actions to provide an acceptable service level and gracefully recover. One then injects a fault to force a particular failure mode and verifies whether the appropriate mitigation actions are indeed triggered.

DevOps. Orthogonal to these four decisions is when to incorporate the fault injection in the build pipeline. For safety-critical systems this is done at the pre-release stage where all failure scenarios are exhaustively tested. As this is prohibitively expensive and may involve a series of manual interventions this doesn't lend itself well to a DevOps approach.

Chaos engineering (cfr. Chaos Monkey below) therefore hand advocates to perform fault injection while the system is in production, as it creates a culture of resilience in the DevOps team.

Digital Twin. The increased trend towards digitalization creates opportunities to perform realistic simulations. A digital twin is a digital version of a manufactured product which is continually updated throughout the physical systems life cycle. By design, a digital twin is a faithful representation of the actual system and as such allows perform realistic simulations of failure scenarios.

1.4.4 Examples (Tools)

Chaos Monkey. One of the better-known examples of fault injection is the Chaos Monkey, a tool invented by Netflix test the resilience of its IT infrastructure. It mimics a monkey entering a data center and randomly ripping cables, destroying devices, and knocking out servers. Chaos Monkey was later on incorporated into the Simian Army, where a plethora of different monkey (i.e. fault types) were included.

Digital Twin.

Serge ►*Example from cyber-physical systems here.*◄

1.4.5 Pros and Cons

Fault injection is possible when the DevOps build pipeline allows to execute tests against the (pre-release version of) the system under test. However, as always it comes with some advantages and disadvantages.

- *Standard Compliance*. Fault injection is one of the techniques which is required by safety critical standards (e.g., ISO 26262 for Automotive). By automating the process, it will be easier to demonstrate that the process is compliant with the standard. Serge ►*Look up these standards*◄
- *Resilience Culture*. Chaos engineering reportedly creates the capacity for dealing with major disruptions 24-7. Just like fire-fighters are trained for interventions under stress conditions, fault injection produces the capacity in the DevOps team to deal with inevitable emergencies.

- *Heavyweight*. To determine what-where-when-how for the faults to be injected a heavy weight risk analysis (e.g., failure mode and effect analysis — FMEA) is required. Establishing a good oracle to express an acceptable service level and a graceful recovery is particularly challenging.
- *Expertise*. Setting up the build infrastructure to inject the faults and observe the systems behavior demands intimate knowledge of the architecture of the system under test.

Study Questions

1. Mutation Testing and Fault Injection both inject artificial defects in the system under test. Yet the purpose to do so is entirely different. Explain briefly what the main differences are.
2. One way of classifying fuzzing tools is the attack surface which is systematically exposed with malformed input. List three commonly used interfaces where fuzz testing is applied.

References

1. Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.C., Laprie, J.C., Martins, E., Powell, D.: Fault injection for dependability validation: a methodology and some applications. IEEE Transactions on Software Engineering **16**(2), 166—182 (1990). DOI 10.1109/32.44380
2. Atlidakis, V., Godefroid, P., Polishchuk, M.: Restler: Stateful rest api fuzzing. In: Proceedsings ICSE 2019 (IEEE/ACM 41st International Conference on Software Engineering), pp. 748–758. IEEE Press, New-York, NY (2019). DOI 10.1109/ICSE.2019.00083
3. Baker, R., Habli, I.: An empirical evaluation of mutation testing for improving the test quality of safety-critical software. IEEE Transactions on Software Engineering **39**(6), 787–805 (2013). DOI 10.1109/TSE.2012.56
4. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at facebook. Communications of the ACM **62**(8), 62–70 (2019). DOI 10.1145/3338112
5. Drysdale, D.: Coverage-guided kernel fuzzing with syzkaller (2016). https://lwn.net/Articles/677764/ — last accessed September 2023
6. Dupuydauby, C.: Mutation testing: first steps with stryker-mutator .net (2019). [on line] https://gist.github.com/dupdob/60fefe8495c8ebe8638ffcffc98a8703 — last accessed In April 2020
7. Eisele, M.C., Maugeri, M., Shriwas, R., Huth, C., Bella, G.: Embedded fuzzing: a review of challenges, tools, and solutions. Cybersecurity **18**(5), 1–18 (2022). DOI 10.1186/s42400-022-00123-y
8. Grieco, G., Ceresa, M., Buiras, P.: Quickfuzz: An automatic random fuzzer for common file formats. In: Proceedings of the 9th International Symposium on Haskell, p. 13–20. Association for Computing Machinery, New York, NY, USA (2016). DOI 10.1145/2976002.2976017
9. Jia, Y., Harman, M.: An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering **37**(5), 649–678 (2011). DOI 10.1109/TSE.2010.62. URL https://doi.org/10.1109/TSE.2010.62
10. Li, N., Offutt, J.: Test Oracle Strategies for Model-Based Testing. IEEE Transactions on Software Engineering **43**(4), 372–395 (2017). DOI 10.1109/TSE.2016.2597136. URL https://doi.org/10.1109/TSE.2016.2597136
11. Manes, V.M., Han, H., Han, C., Cha, S., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering **47**(11), 2312–2331 (2021). DOI 10.1109/TSE.2019.2946563
12. Offutt, A.J., Untch, R.H.: Mutation 2000: Uniting the Orthogonal. In: W.E. Wong (ed.) Mutation Testing for the New Century, pp. 34–44. Springer US, Boston, MA (2001). DOI 10.1007/978-1-4757-5939-6_7. URL https://doi.org/10.1007/978-1-4757-5939-6_7
13. OSSFuzz: OSS-Fuzz: Continuous fuzzing for open source software (2023). https://github.com/google/oss-fuzz — last accessed September 2023
14. Petrović, G., Ivanković, M.: State of mutation testing at google. In: Proceedings ICSE2018 (40th International Conference on Software Engineering: Software Engineering in Practice), pp. 163–171. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3183519.3183521
15. Ramler, R., Wetzlmaier, T., Klammer, C.: An empirical study on the application of mutation testing for a safety-critical industrial software system. In: Proceedings SAC2017 (Symposium on Applied Computing), pp. 1401 – 1408. Association for Computing Machinery, New York, NY, USA (2017). DOI 10.1145/3019612.3019830
16. Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E., Winter, C.: Tricorder: Building a program analysis ecosystem. In: Proceedings ICSE2015 (37th International Conference on Software Engineering), pp. 598 – 608. IEEE Press (2015)
17. SQLite: How SQLite is tested (2023). https://www.sqlite.org/testing.html — last accessed September 2023
18. Werquin, T., Hubrechtsen, M., Thangarajan, A., Piessens, F., Mühlberg, J.T.: Automated fuzzing of automotive control units. In: Proceedings SIOT 2019 (International Workshop on Secure Internet of Things, pp. 1–8. IEEE Press, New-York, NY (2019). DOI 10.1109/SIOT48044.2019.9637090

19. Zou, Y.H., Bai, J.J., Zhou, J., Tan, J., Qin, C., Hu, S.M.: TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing. In: 2021 USENIX Annual Technical Conference (USENIX ATC 21), pp. 489–502. USENIX Association, Berkeley, CA (2021)