J'ai fait du system testing en quelque sorte puisque j'ai vérifié que le programme fasse le boulot qui lui est demandé de faire avec des inputs raisonnables.

**Non-Functional Input** : Inputs that have nothing to do with the API provided by the software that we are testing and nothing to do with the API that are used by the software that we are testing.

**White Box Testing** : Structural tests focus on the implementation with knowledge of the program.

**Black Box Testing** : Functional tests focused on the specication.User perspective without knowledge

**Unit Testing** : Testing a module in an isolation fashion.

**Integration Testing** : Taking multiple software modules that have already unit tested in combination with each other.

**System Testing** : Test if the system as a whole meets his goal

**Differential Testing** :Testing the same input across dierent implementations of the software under test and comparing them for equality.

**Stress Testing:** Test at or beyond the normal limits of a software .

**Random Testing**: Using the result of a random number generator to randomly create test inputs and deliver them to the software under test.

CHAP 2

**Function Coverage** Every function in the code is executed during testing.

**Statement Coverage** Every statement in the code is executed during testing.

**Line Coverage** Every physical line in the code is executed during testing.

**Branch Coverage** Every branch in the code is executed in both way during testing.

**Loop Coverage** Every loop in the code is executed 0 times, once, and more than once during testing.

**Modified Condition/Decision Coverage** It's an hybrid which use branch coverage techniques and must takes on every possible input. Furthermore, every condition independently must aect the outcome of a decision.

**Path Coverage** Every path in the code is executed during testing.

**Boundary Value Coverage** Each boundary value in the code must be checked during testing.

**Synchronisation Coverage** Ensure that during testing the locks actually does something.

CHAP 3

**Oracle** : Determine whether the output of the software under test is either good or bad.

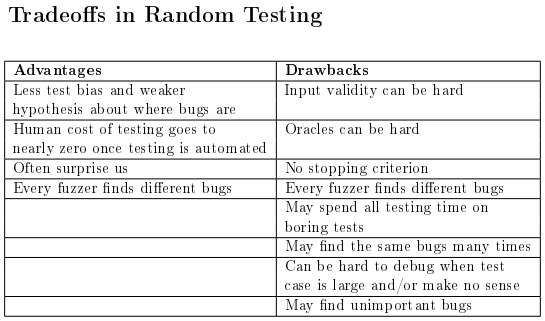
**Fuzzing** : Automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program

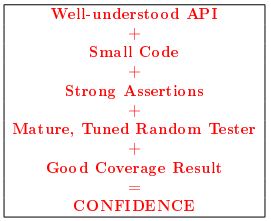
**Testcase Reduction** : Piece of technology that's very often combined with *random testing*. It takes a large test case that make the software under test fail and turns into smaller test case that still make the software under test fail.

**Weak Oracles** Detect whether or not an application crashed. Which means that the software under test violated some rule that the hardware, a programming language or an enhanced execution environment imposed.

**Medium Oracles** Assertion checks that the programmer has put into the software. It provide a more specic application kind of checking than does the weak oracles.

**Strong Oracles** Alternate implementation of the same specication 10 , function inverse pair 11 and null space transformation.

Even if random tester is mostly generating stupid test, if it can generate a clever test case one in a million time it still more effective than testing resources by hand.



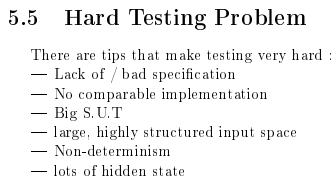
**Bug Triage** : Process by which the severity of different bugs is determined and we start to disambiguate between different bugs in order to basically try to get a handle on which bugs we can report in parallel.

**Core Dump** : Recorded state of the working memory of a computer program at a specifc time, generally when the program has crashed or otherwise terminated abnormally

**Delta Debugging** : Framework that takes a script and takes the test input and automates the process of reduction in a loop which terminates when the delta debugger, which has a bunch of heuristics built in for eliminating parts of the input, can't reduce the input any more.

**Test Suite** : Collection of test which often can be test automatically and that we run periodically

**Regression Test** : Any input that cased some previous version to fail



FUZZER:

J’ai fait du system testing en vérifiant bien que le fait de modifier les marqueurs de début et de fin d’image ne faisait absolument jamais crash les applications.

J’ai fait du random testing car j’ai utilisé à plusieurs occasions un générateur de nombres aléatoires pour créer des inputs et les donner au programme qui est testé.

J’ai fait du mutation-based random testing car je pars d’un fichier qui est déjà existant/créé, et je modifie les bytes pour tester le programme.

La modification des bytes se fait de deux manières:

-La première se fait avec des nombres totalement aléatoires entre 0 et 256. C’est donc une forme de black box testing car on n’utilise aucune connaissance, on se contente simplement de modifier le fichier n’importe comment en espérant générer une erreur.

-La deuxième pioche dans une liste de nombres qui sont connus pour générer des erreurs assez souvent dans les programmes. Typiquement des erreurs de taille de type de données et des erreurs arithmétiques. On peut donc considérer que c’est une forme de white box testing car on ne modifie plus les bytes à l’aveugle. On sait que ces valeurs ont de grandes chances de faire planter le programme et donc on les utilise abondemment à différents endroits dans le fichier.

J’ai essayé de faire du generative random testing en utilisant un header et footer de fichier jpg au milieu desquels j’ai inséré des bytes aléatoires mais je n’ai pas réussi à obtenir un seul fichier qui me donnait une image valide donc j’ai abandonné cette idée.

A FAIRE : code coverage.

Pourquoi éviter les marqueurs SOI et EOI?

- Car on ne veut pas changer le type de fichier, on veut garder le même pour qu’il puisse être pris en compte par le programme et générer des erreurs.

Ne pas oublier: il faudra voir dans les logs que SOI et EOI ne font pas crash, et qu’il faut donc éviter de les insérer dans les bytes.