Endless Science Open Journal of Software Engineering

Software Testing or The Bugs' Nightmare

Héctor D. Menéndez[*]

Middlesex University London and Endless Science

h.menendez@mdx.ac.uk

Abstract

Software development is not error-free. For decades, bugs –including physical ones– have become a significant development problem requiring major maintenance efforts. Even in some cases, solving bugs led to increment them. One of the main reasons for bug's prominence is their ability to hide. Finding them is difficult and costly in terms of time and resources. However, software testing made significant progress identifying them by using different strategies that combine knowledge from every single part of the program. This paper humbly reviews some different approaches from software testing that discover bugs automatically and presents some different state-of-the-art methods and tools currently used in this area. It covers three testing strategies: search-based methods, symbolic execution, and fuzzers. It also provides some income about the application of diversity in these areas, and common and future challenges on automatic test generation that still need to be addressed.

Keywords: Software Testing; Bugs; Test Generation; Symbolic Execution; Fuzzing; Search-based Testing; Diversity

1 Introduction

Although there are different ways to understand programs, we can always start considering the two most basic levels of abstraction: syntax and semantics. Software's syntax corresponds with the language or languages that define the whole program while its semantics corresponds with the logic behind its purpose. These are the two main places where we normally look for bugs [50]. These bugs are normally either syntactic or semantic errors that affect the behaviour of the program. Although other components, such as the operating system or the hardware, produce bugs in programs, we tend to

[*]Corresponding Author

focus on isolating the specific part of the program's code or logic to locate the error behind the bug.

Several methods are aiming to identify these problems. For instance, verification extracts the logic of the program and evaluates whether it is correct [80]. Another option is debugging: execute the program step by step trying to discover in which part of the program the error manifests [88]. If the error is related to over-consumption of the system's resources, a common strategy is to use profiling, which provides information about the memory and time usage of every part of the program [10]. Nevertheless, if we can understand a program as a function and focus on its input/output behaviour, i.e. check that the output of a specific input is correct, we can apply software testing [63].

Software testing is partially a science and also an art that aims to generalize the identification of every single bug in a program. It first starts with different granularity levels, depending on whether a developer or tester aims to test specific parts –or methods– of the program or the program as a whole. These are respectively unit and system testing [4]. Furthermore, it is important to consider whether the program keeps its retro-compatibility or new parts integrate properly with the rest of the program. These fields are respectively regression [55] and integration testing [45]. It is also relevant to know whether the goal is to test, for instance, the security of a program (for example, with penetration testing [8]), the way the program threads interact (concurrency testing [81]) or whether its logic follows a specific model (model testing [28]). Although there are several ways and perspectives to understand software testing, this humble document focuses concretely on the methodologies for automatic test generation [65].

This introductory paper does not aim to be a complete survey about software testing but an anecdotal walk through different tools and methods that I have found during my research. That is the reason it is mainly focused on search-based strategies –including fuzzers–, which was the main focus of my research group (CREST) during my work at UCL, and all our amazing collaborators that used to attend the CREST Open Workshops (COWs) or the InfoGang meetings; Another relevant part of the paper focuses on symbolic and concolic execution, which was the main field of several people I had the honour to meet during different research talks, mainly organized by Philippa Gardner and the VETSS institute[1], normally at Microsoft Research at Cambridge.

The rest of the paper continues with a simple introduction to automatic test generation (Section 2), then it introduces the three main techniques I aim to discuss: search-based testing (Section 3), symbolic execution (Section 4), and fuzzing (Section 5). After, it discusses the relevance of diversity methods in automatic test generation (Section 6) and, before the conclusions, it provides a series of challenges that automatic test generation is facing or will face (Section 7).

2 Automatic Test Generation

Testing starts with a program P that accepts inputs in the space of inputs I and generate outputs in the domain O. If the program is deterministic, every input $i \in I$ will generate a unique output $o \in O$, and this output will satisfy $o = P[i]$, i.e. every output is the solution of applying the program to a specific input. Moreover, deterministic programs also satisfy that $O = P[I]$, i.e., the output domain is generated by the input

[1]https://vetss.org.uk/

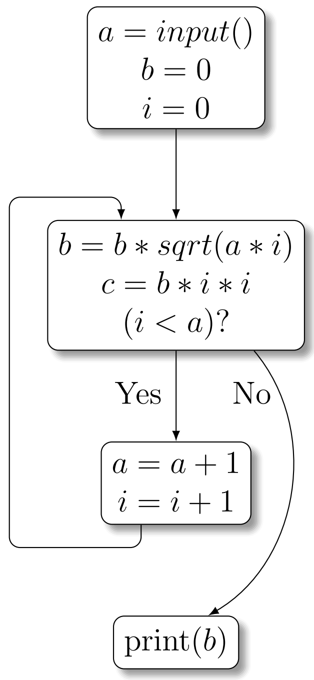

Figure 1: Example of a program divided into four basic blocks. The first one collects the inputs and initializes the variable. The second is the body of a loop whose condition is verified at the end. The third one is the increment of the loop, and the last one is a simple print at the end of the program.

domain. Testing every possible input and verifying the correctness of every possible output would show whether the program is correct or not, however, the input space is normally immeasurable and the execution costs can also be too high for testing every possible input.

This led a different strategy: if we can not test every input, can we, at least, test every behaviour? But, what is *a behaviour*? Testing semantics is always a difficult challenge because it requires knowledge about the program and how much it satisfies its requirements; therefore, the idea of behaviour needs to be a flexible concept that we can concretize and generalize to every program [77]. Defining a behaviour as a walk inside of the program's control flow graph makes any behaviour into a program path [40]. If we cover every path, we reach maximum program coverage concerning these behaviours.

A program can be syntactically divided into basic blocks (Figure 1 shows an example of a program represented as a set of basic blocks). Each block satisfies that all the code inside the block will run together. The blocks are connected by conditional and unconditional jumps, normally as consequences of loops and branches in the code syntax (the connections correspond with the arrows of Figure 1). The connections among the blocks create the control flow graph, representing the program's paths from its inputs to the outputs. Every path (i.e. set of edges) inside the graph starting from the input block and leading to the output one is a program path. Different inputs can traverse the same path in a deterministic program, reducing then the testing possibilities.

Although the idea of targeting paths as a testing goal is an improvement to testing every single input, it is not a solution. Programs contain loops and if, for example, the limit of a loop is controlled by an input, this would create as many possible paths for that program as the size of that input. Assuming that this can happen with several loops inside the program, testing by behaviour is also a complex task. This specific phenomenon where paths grow exponentially with respect to the inputs is called *path explosion* [85]. Testers need to reduce the notion of coverage even more, so they go back to the control flow graph.

The edges of the control flow graph are good candidates for coverage. This coverage strategy is called branch coverage [40]. The main goal of this testing strategy is to cover every single branch. Although this is an under-approximation of the whole set of behaviours of a program, which is every path, following the previous notion, it is a good compromise between effectiveness (i.e., activating as many behaviours as possible) and efficiency (i.e. time consumed during the testing process), and it becomes one of the most demanding notions of coverage, normally in tandem with line coverage or other coverage criteria [71].

Having a notion of coverage provides the tester with a goal, and this goal allows to measure the progress of the testing process. This ability to measure progress gives us a framework to automate the process of testing and develop different automatic testing methods that can help to find bugs inside of programs. And also, remember: "If you can measure it, you can improve it".

3 Search-Based Methods

Search-based methods for testing mainly leverage different bio-inspired strategies to improve coverage. The most prominent search-based strategy is evolutionary algorithms, one of the main branches of evolutionary computation [78].

The simplest way to define an evolutionary algorithm is the evolutionary process of chromosomes. Starting with a population of chromosomes, there are three main steps that these algorithms follow: 1) the chromosomes are selected for reproduction according to one or more fitness criteria, 2) they cross to generate new individuals, and 3) the new individuals suffer random mutations. This simple process has proven to be an amazing methodology for optimization problems [38]. There are several ways to apply it, depending on the domain where they need to work, but the most known are:

- Genetic algorithm [78]: this one is characterised by optimising only one specific objective, and using a single population of chromosomes to perform the optimization.
- Evolutionary strategy [25]: there is only a single chromosome that is optimized, therefore there is no crossover and the mutation is normally adaptive.
- Multi-objective genetic algorithms (MOGA) [26]: the optimization process has several goals, normally defined as fitness objectives. Due to there is not a clear solution as the optimization of one objective might affect negatively the optimization of another one, these kinds of algorithms define a frontier of solutions, called the Pareto front, which shows a trade-off among the different objectives.
- Co-evolution [57]: in this case, more than one population either compete or collaborate during the optimization process. This normally relates to game-theory. For instance, the most common competition scenario is a zero-sum game where two populations compete and the profit of one is the loss of the other.

Beyond evolutionary computation, other search strategies are used for software testing, for example, inside bio-inspired methods, we find strategies based on ant colony

Technique	Search-strategy	Application
EvoSuite (WTS) [30]	Genetic Algorithm	General Unit Testing
EvoSuite (DynaMOSA) [68]	Multi-Objective	General Unit Testing
EvoSuite (MOSA) [67]	Multi-Objective	General Unit Testing
EvoSuite (MIO) [7]	Multi-Objective	General Unit Testing
Dorylus [18]	Ant Colony Optimization	General Unit Testing and Object Sequences.
Mathematical Execution [32]	Monte-Carlo Methods	Unit Testing mainly numerical.
AFL [87]	Genetic Algorithm (Fuzzer)	System Testing
AFLFast [14]	Genetic Algorithm (Fuzzer)	System Testing
FairFuzz [54]	Genetic Algorithm (Fuzzer)	System Testing
Libfuzzer [74]	Genetic Algorithm (Fuzzer)	Unit/System Testing

Table 1: Some of the search strategies mentioned during this work and their applications.

optimization [27] (ACO), while, related to other statistical approaches, we can find methods based on Monte Carlo search trees [16].

Keeping this in mind, when we study the literature, we can define a good taxonomy of different techniques based on the search strategy they use. Although this paper does not aim to define this taxonomy, Table 1 shows a simple schema of some relevant techniques, from the literature mentioned during the paper, where the different strategies are related.

Some examples of different testing strategies based on genetic algorithms come from EvoSuite [29], a popular automatic test generation tool for Java (Section 3.1). EvoSuite employs algorithms mainly based on MOGAs. In the case of ACO, some tools like Dorylus [18] leverage this paradigm to create inputs for programs, even in the context of object-oriented programming where sequences of object's methods need to run in a specific order [19]. For Monte Carlo methods, tools like CoverMe or mathematical execution have proven to detect bugs even in the context of floating-point numbers [32].

3.1 Popular Algorithms

The design of search-based methods normally starts with an encoding representing the test suite and a fitness function representing the test goal, normally coverage (either line, branch or path coverage) [29]. This simple kind of algorithm will attempt to create the smallest test suite that reaches maximum fitness, as it is normally called the whole test suite algorithm and it is the most basic algorithm for one of the most popular unit test suite tools, called EvoSuite [29]. EvoSuite is a testing framework for Java with more than 10 years of development. It is very popular in the community and has been able to compete with several commercial and state of the art tools for years [31]. Different authors leverage the main framework of EvoSuite to develop search-based algorithms, where the most popular are Whole Test Suite generation (WTS) [30], the Many-Objective Sorting Algorithm (MOSA) [67], the Multiple Independent Objective (MIO) algorithm [7] and Many-Objective Sorting Algorithm with Dynamic target selection (DynaMOSA) [68].

These four algorithms cover the best examples of the main paradigms for search-based automatic software testing. WTS generates the test suite based on the idea that not all the coverage goals are similarly difficult to reach. It generates the whole test suite as individual solutions of an evolutionary search. To reach all the branches, it uses two metrics to create a gradient during the search process: approach level and

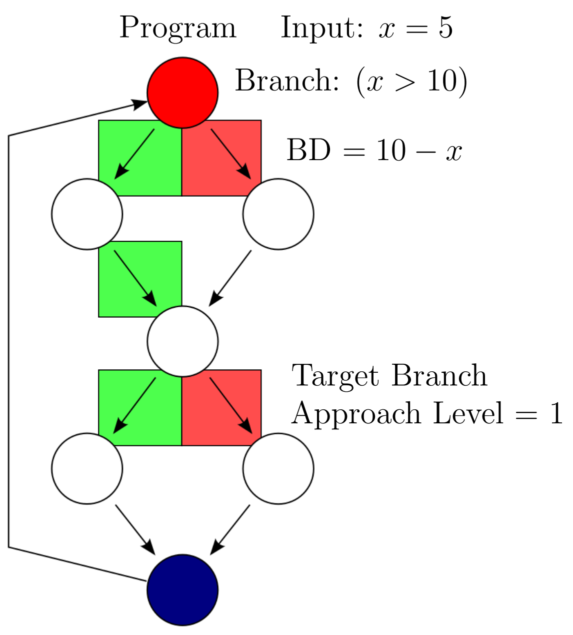

Figure 2: Control flow graph example with approach level and branch distance metrics. Reprinted and modified from JulesH, by Wikipedia Commons, February 13 2021, retrieved from `https://commons.wikimedia.org/wiki/File:Control_flow_graph_of_function_with_two_if_else_statements.svg`.

branch distance. These metrics measure how close the algorithm is to cover the whole program. The approach level calculates how many branches apart the input is from the target branch. Branch distance makes the approach level smoother. When an input is close to a branch and needs to traverse it, it says how far the input is to traverse that branch, in terms of the branch's comparison operation. In the example of Figure 2, we can see that the input $x = 5$ traverses the first condition $(x > 10)$ through the false branch. If we calculate the branch distance, it is $10 - x$ because it is how much x needs to increase to do the condition true. Concerning the approach level, if we consider any of the figure's red branches as the target one, the approach level is 1 in both cases, because only one branch needs to be flipped to reach the target.

In MOSA each testing target is an objective to optimize. When a target is covered, the test covering it is stored in an archive. For each uncovered testing target, the best individual gets the best rank (in terms of fitness). Both MOSA and WTS suffer four main problems [7]: both give more emphasis to exploration, which is more expensive in constrained situations such as system-level system; they keep good individuals of covered targets inside of the population, which affects the search quality; when an infeasible target creates an individual with high fitness, they tend to lead the population; and when the number of objectives is large, the population size needs to be able to respond in contrast, which is a trade-off.

To solve the problems related to WTS and MOSA, Arcuri [7] presented the Many Independent Objective algorithm. MIO separates each target into different search problems, keeping an archive of those tests that are covering each target. At the same time, it keeps a population of tests per target. In every iteration of the search process, MIO selects a test uniformly at random from the archive, runs it, and mutates it. If the test is good for specific targets, it adds the test to the archive of those targets. The main difference to the previous algorithms is that, in this case, there is not a general search for all the targets but an independent one per target with an associated population, therefore it is more dynamic and adapts better to the program.

DynaMOSA [68] is an extension of MOSA that aims to improve the selection of targets during the search process. The system chooses the targets dynamically, dividing them into sets. It focuses on sets of uncovered targets and creates a hierarchy among them, called the control dependency hierarchy. The higher in the hierarchy the higher the covering priority. This increases the diversity of the coverage process, especially at the beginning of the search. Once a lead uncovered target is covered, it is removed and the next one in the hierarchy takes its position. This makes the search more efficient under time restrictions.

Although there are other interesting algorithms related to search-based strategies, the following introduces some interesting out-of-the-box applications that extend from these paradigms.

3.2 Out-Of-The-Box Applications

Search-based testing methods are a strong tool to validate some specific behaviours of software, and, for that reason, they also became the base for more sophisticated methods, such as automatic software repair [52], genetic improvement [69] and software transplantation [11].

These three applications cover different aspects where the software either needs fixing or improving. For example, automatic software repair performs transformations on the implementation at the syntax level intending to fix known bugs automatically. The program uses the test suite and their assertions as a fitness function and applies mutations to the program's abstract syntax tree to find one that passes the tests correctly. Although this idea proved to solve significant errors in software [53], there are some limitations. The three main limitations are related to those code changes that the tool can learn [56], over-fitted patches that pass with the tests but do not fix the bug [51], and the scalability of the technique when it deals with large programs [34].

When we focus on the non-functional aspect of the software, such as energy, memory or time consumption, we can apply genetic improvements [69]. Genetic improvement is also based on the idea of combining genetic programming and test suites, however, instead of fixing the program, as automatic program repair, genetic improvement uses the test suite as a sanity check during the non-functional optimization process. The evolutionary process will manipulate the program syntax to improve a non-functional requirement while the semantics (verified by the test suite) remains [69]. This process has successfully optimised software in different ways, some examples are energy [17] and time consumption [49]. It also has already opened new areas of research such as adversarial genetic programming for cybersecurity [66]. However, the main limitation of genetic improvement is code readability, which is one of the open problems of the area [1].

The last extension of genetic improvement, which combines both functional and non-function improvements, is automatic software transplantation [11]. The main idea is to identify a program, for example, a video player, that requires a new feature, for example, a video codec. Then, we need to identify a donor with that specific feature and the implantation point into the host, and the automatic transplantation system will create the "fitting" to make that feature part of the host. This is performed via testing and genetic improvement. Mainly, the transplantation system modifies the organ until it fits in the implantation point based on the response to the test suite.

Although these are just a few out-of-the-box applications of search-based testing, there are several more that are under development, and their main goal is to provide better software in functional and non-functional ways.

4 Symbolic Execution

Symbolic execution is one of the main paradigms for the verification of programs. It aims to create a semantic interpretation of the program's implementation to validate it [46]. The main idea behind symbolic execution is the identification of paths and values for these paths. Imagine the example of the basic blocks in Figure 1. Let e be the program's entry and n_i each block. Then, a path $e \to n_1 \to \cdots \to n_k$ denotes the blocks that a program execution traverse (or a trace). In symbolic execution, for each path (for example, $e \to n_1 \to \cdots \to n_k$), we define a symbolic state $\{(v, \phi_v)\}$ composed by both a symbolic expression (ϕ_v) and the variables of that expression (v satisfying $v \in Var(P)$). The bottom of Figure 3 shows the symbolic expressions for the variables a and b. In this case, they are directly assigned to the inputs. Apart from the symbolic state, we also define the program constraints ($c_1 \wedge \cdots \wedge c_q$), which is a set of constraints associated with that path. In the figure, the constraints correspond to the `guard` statements.

To formalize it, the path $e \to n_1 \to \cdots \to n_k$ from a program P defines the symbolic state as the triple:

$$(e \to n_1 \to \cdots \to n_k, \{(v, \phi_v)\}_{v \in Var(P)}, c_1 \wedge \cdots \wedge c_q)$$

To generate test inputs using this methodology for a specific program, the program needs to be transformed into a set of constraints based on the input (as Figure 3 shows). This is normally performed using Static Single Assignment (SSA) [37]. Initially, the program's expressions are translated into single static assignment and their loops are unwound up to a specific level. Unwinding reduces the number of constraints related to the loops, creating an under-approximation of the program's semantics or an over-approximation of the input set, i.e., some inputs might not reach the specific program path. Symbolic execution normally leverages ϕ-functions to propagate values when branches join and it also simplifies expression using constant propagation [48]. This aims also to remove infeasible branches. The verification conditions become a formula describing the program f_P, formatted as a set of program constraints. Each constraint is a control flow instruction that works as an expression based on the input values.

This can also be particularized to focus on testing, where we want to generate inputs that reach a specific program point. In this case, given the program P, and a program point $pp \in P$, focused testing leverages symbolic execution to generate an invalid assert condition immediately before pp. Angeletti et al. [6], introduced this process by using the C-Bounded Model Checker (CBMC) [48] to generate the program constraints that guide the tests. Following the example of Figure 3, this method just adds an `assert(0)` statement in line 6, after the `printf` expression, still inside the `if` then-block. CBMC triggers a verification error at this point, providing verification conditions for it and distinguishing every possible path traversing that point. It creates these verification conditions using symbolic execution. These verification conditions are

Original Program

```c
#include<stdio.h>
int main(int argc, char *argv[])
{
  int a,b,c,d;
  scanf("%d %d",&a,&b);
  if(a>b) c=a;
  else c=b;
  d=(7-c)/6;
  if(c==6) printf("0/1\n"); //pp
  else if(c==1) printf("1/1\n");
  return 0;
}
```

Symbolic State for pp

```
1 And(guard_1 == (b < a),
2     guard_2 == (If(guard_1,a,b) == 6),
3     guard_2,
4     a == input_0,
5     b == input_1)
```

Figure 3: Example of a program and the expression tree for the program point *pp* highlighted in the original program.

normally in SMT-LIB version 2 format [12], so an SMT-Solver like Z3 [24] can be used to generate witnesses for them, which will be test inputs traversing that point.

Figure 3 (bottom) shows an example of the expression needed to reach *pp* in Figure 3 (top). The constraints are defined as a tree rooted at an `And` expression. The tree is divided into three parts: definition of guards (line 1 defines guard_1; line 2 defines guard_2), input definitions (line 4 for `a` and line 5 for `b`), and path control expressions (line 3 activates guard_2). It is relevant to remark that, in this example, only guard_2 needs to be activated to reach *pp*, but since its variable, `c`, depends on guard_1, this guard must be also kept. The simplification process removes any other guard because they are not related to *pp*.

One of the main limitations of symbolic execution is path explosion [85]. This normally happens when the program has several paths, especially when there are many nested loops involved, and the engine is not able to generate constraints for all of them. To alleviate this problem, some researchers developed a technique, named concolic testing [72], that combines concrete and symbolic execution. The main idea is to create a concrete input and set constraints that help to traverse those paths that that input was not able to traverse. Instead of creating a symbolic representation of the whole program, only those inputs related to uncovered paths are considered.

4.1 Popular Tools

Symbolic execution engines normally have to deal with three problems [20]: path explosion, constraint solving and memory modelling. These three problems normally define different strategies when symbolic execution tools are designed. Some popular tools leverage this paradigm to automatically create test suites, mainly for C or Java programs. Some of the main examples are KLEE [21] and EXE [22]. Also, there are tools based on concolic execution such as DART [35], CUTE [73] or JDart [58] or even more modern techniques such as CATE [60] for Android.

One of the areas that sometimes competes with –and other times complements– symbolic execution is model checking. In model checking a specification is provided along with the software and the model checking engine makes sure that that specific requirement is satisfied [48]. For that, it creates a model of the program using propositional logic (similarly to the program formula mentioned in Section 4) and the specification is applied to the solvers to verify it. Several tools of model checking either use or are extended to symbolic execution engines. Some examples are the C-Bounded Model Checker (CBMC) [48], which uses symbolic execution to create a test suite for the program it aims to analyse, or Java PathFinder [41], which has a specific symbolic execution extension [5].

5 Fuzzers

Recently, a popular testing methodology is raising, named fuzzing. The main goal of fuzzing is to cover the whole program by generating different mutations of a provided set of inputs (or seeds) [89]. Fuzzing has the advantage of scalability, which is the main skill of the most popular fuzzing tool: the American Fuzzy Lop (AFL) [87].

During the last few years, fuzzers are becoming dominant tools in testing. They exposed multiple new bugs in several all-day systems, no matter their scale. A good example of the abilities of fuzzing is that AFL exposes the Heartbleed bug from OpenSSL in about 10 minutes [15]. This is surpassing the current state of the art in testing [47].

The main strategies in fuzzing are black and grey-box, although, in the fuzzing terminology, they are normally described as those based on generation and those based on mutation, respectively [47]. In generation fuzzing, the fuzzer receive no feedback from the program under test. It generates inputs either randomly or following a specific grammar to identify potential vulnerabilities within the program [43]. Some good examples of fuzzers following this strategy are Zzuf [43] or Radamsa [42]. For mutation-based fuzzers (or feedback-based), they instrument the program and receive feedback related to new paths, branches or crashes discovered by the last input [47]. AFL is the most popular fuzzer following this strategy and it has several extensions, such as AFLFast [14] or FairFuzz [54]. Other tools either work similarly, like Libfuzzer [74], or extend its abilities, like QSYM [86].

Although fuzzers can be understood as search-based strategies, especially when they are mutation-based, the two communities normally tend to separate their main targets. Normally the main target of search-based algorithms is the oracle problem, related to the functionality of the software, while the main target of fuzzers is exposing crashes and sometimes exploitable vulnerabilities. Two good tools that normally work in tandem to expose vulnerabilities are the AddressSanitizer [75] and crashwalk [64], a triage system for vulnerabilities.

5.1 Popular Tools

Fuzzing started mainly as a black-box approach for system testing where different strategies were applied to a set of inputs. Examples of tools from these times were zzuf [43] and Radamsa [42]. The former, Zzuf, considers the input in its binary form and mutates some bits to generate new ones. These modifications are deterministic. The latter, Radamsa, is similar to Zzuf although it includes different heuristic and operates with different types of inputs such as network ones.

AFL [87] led the evolution to feedback-based or mutation-based fuzzers. AFL instruments the whole program (normally in C or C++ during compilation time by using either Clang or GCC) and uses this instrumentation as a guidance that shows which pair of branches (or transitions) have already been covered. Starting with a set of seeds, it queues them. It will mutate the elements of this queue through heuristics to discover new transitions. If a mutated seed discovers a new transition, it will be added at the end of the queue to continue with the mutation process. Two popular extensions of AFL are AFLFast [14] and FairFuzz [54], whose aim is to maximize the exposition of rare paths. AFLFast has a scheduler controlling how many times a seed has been mutated. This helps to select the next seed from the queue that will be mutated. FairFuzz goes a step further, it does not only smartly select the seeds, but it also selects the heuristics that will mutate these seeds. Another popular tool, in this case, embedded into the LLVM compiler infrastructure, is Libfuzzer [74]. This library generates inputs for a specific function (or target), which becomes the entry-point of the program.

Some tools extend the abilities of fuzzers. For example, QSYM combines fuzzing with concolic execution to reach those targets whose branch conditions are difficult for the heuristics [86]. Also, some fuzzers combine with grammars to improve the generation of structured inputs. Some examples of these fuzzers are Gramfuzz [39], which uses the input's grammar to construct structured trees based on them and mutate the trees directly to preserve the structure, and Superion [82], which constructs on top of the previous idea by generating two new mutation strategies, one based on sub-trees and the other one on dictionaries. In contrast with grammar-based approaches, fuzzers like AFL use dictionaries during the input generation process.

For those interested in fuzzing datasets, Google provides a service named the continues fuzzing services for open source software or OSS-Fuzz [76] and also a benchmark system to compare new fuzzing implementations named FuzzBench [62]. Also, to perform a proper comparison with the state-of-the-art in fuzzing, Klees et al. collected a set of good practices in fuzzing with pieces of advice related to formal comparisons [47].

6 Diversity methods

Testing processes aim to provide a comprehensive exploration of the behavioural space of a program to detect potential errors in the program's logic or bugs [63]. Although this idea would immediately lead to the notion of diversity, considering that testing is not infinite and will need to stop at some point, multiple factors limit this exploration, and different techniques have been introduced to compensate for these problems.

One of the first things that diversity-based testing aims to define is the notion of diversity. In terms of our previous formalization (Section 2), we can define the diversity of inputs (I), outputs (O) or behaviours (B). The goal is either to maximize the diversity or increment it, but the definition of diversity can lead to a long debate in different areas. If we consider the domains of spaces for I, O and B, for a deterministic program, we can consider that $|I| \geq |B| \geq |O|$ (i.e. there are more inputs than behaviours and more behaviours than outputs). It is not possible to have more inputs than potential behaviours, because one input leads to one behaviour, considering the notion of a path as behaviour and, equivalently, to outputs. However, this information loss between inputs, behaviours and outputs makes diversity non-transitive.

Some of the definitions will consider diversity as spread within any of these domains.

The more areas of the space that are cover, the better [23]. Others would define it in terms of probability. In this case, we can define the tester as a generator G and aim to calculate how close this generator is to a specific probability distribution, for example, the uniform distribution where every element has a probability of $1/|D|$ where D is the domain. In this case, our generator would generate a different input, output or behaviour every time. This is kind of obvious in terms of input-based generation because it is simple to improve diversity in terms of inputs. But behaviours and outputs are not so simple [61]. Normally, to reach these goals we either need a semantic notion of the program or to discard inputs when they are not increasing the diversity [3]. Some examples of these are the different approaches to output diversity, for example, output uniqueness [3] generates multiple inputs and discards those that generate the same output. On the other hand, Reza et al. use Simulink to generate outputs and define a notion of similarity in the output space to discard similar outputs [59]. In contrast with these two, output diversity driven creates a generator that generates inputs whose output follows a near-uniform distribution, eliminating the need for discarding inputs [61].

Some authors also try to add diversity directly inside of symbolic execution. A good example was introduced by Gotlieb et al. [36], where the authors generate a uniform testing tool that was based on symbolic execution. Also, the work of Chakabordy on r-wise independent hash functions goes in this direction [23]. In this work, the authors consider the sampling process of the solver bias in terms of diversity. To avoid this bias, they include new constraints in the program's function and force the solver to sample from different domain regions through these new constraints. This increase the spreads and improves diversity.

7 Common Problems and Future Challenges

Although these techniques are good for detecting bugs there are several problems in testing that need a deeper understanding. One of the main problems is scalability [90]. Some of the techniques that allow a deeper understanding of the program, like symbolic execution (Section 4), do not allow to scale to large programs easily (without the extra help of concolic execution), while other techniques that scale, like fuzzers (Section 5), require a deeper understanding or how to construct better inputs to deal with difficult cases.

Another relevant testing problem is multi-lingual programs, where the instrumentation might reach some parts of the program but not others. Multi-lingual programs very common in several projects (Github is a good example of it [83]) and is becoming an interesting challenge in different testing disciplines. For instance, researchers are trying to combine formal languages for SQL and C to perform complete coverage [2].

Following the previous point, normally different languages apply to different cloud services, and the combination of these services also needs to be tested. This is called cloud-testing and focuses on both: testing as a cloud-service and testing apps in the cloud [33]. Considering that the internet is tending to the cloud, it is important to be able to reach all these services during the testing process.

From a theoretical point of view, one of the most relevant problems in testing is when to stop, as Marcel Böhme stated in his STADS framework [13]. Having a notion about the total percentage of coverage discovered during the testing process and a prediction of how much effort is required to discover new paths or bugs will allow the tester to decide when to stop, especially considering that processes like fuzzing normally last days.

Testing concurrent programs is becoming a relevant topic considering that programs leverage the multicore nature of current technologies. Concurrency is one of the limitations of symbolic execution that model checking aims to solve [48], however, modelling concurrency also requires modelling potential non-determinism during the program execution that depends on how the multiple threads are organised by the operating system [9]. One of the main goals during concurrency testing is to achieve high synchronization coverage among the different threads [44]. For that, normally threads are scheduled and the testing process aims to cover all the potential scheduled pairs of threads in a similar way that branches are covered in traditional testing. However, although several approaches have been tried in this area, there is still a lot of work to do to provide complete coverage of concurrent programs [9].

Another interesting testing focus comes with artificial intelligence and machine learning. Apart from those techniques that use machine learning to test program, which were recently collected by Zhang et al. [91], there are researchers developing testing techniques to test machine learning algorithms, either on general machine learning, such as [84], or for the specifics of deep learning implementations, where the neurons of the neural net are the coverage goal [79]. Riccio et al. recently collected multiple different approaches and methodologies for machine learning testing [70]. When testing machine learning, there are two main goals during the testing process: explore the non-determinism of a machine learning system and expose specific adversarial samples that can spoil the classification of the system and might be a threat for future applications of this technology.

8 Conclusions

Automatic software testing is a wide field of research with several different branches that sometimes compete or cross between them. Methods like symbolic execution or search-based algorithms can normally complement each other to achieve better results when bugs need to be exposed. Also, when coverage is not the only goal but a baseline for testing, diversity becomes the best complement to identify potential inputs that might activate unknown bugs, considering its ability to improve the exploration process in adversarial situations as those generated by solvers.

From a performance perspective, fuzzers are gaining major importance not just for their ability to test programs in a system testing way but also because of their performance. They are efficient and they are currently evolving to include techniques related not just to search but also symbolic execution.

Finally, the future challenges of testing will deal with scalability, which is the current target of fuzzers; multiple languages or services, which is the way the cloud is designed these days; and non-deterministic scenarios, as those produced during concurrency or when machine learning or artificial intelligent systems are under test.

Acknowledgements

I want to dedicate this paper to the person, mentor and friend who opened my eyes to the testing world: Dr David J. Clark. David hired me via the InfoTestSS project (EPSRC EP/P005888/1) that I also want to acknowledge. Finally, I want to acknowledge the CREST group where I had hundreds of discussions and coffees about research,

including interesting reading groups, open workshops and several amazing research activities, like our ICSE Fridays. If you, reader, have the opportunity to spend time there, trust me, you won't reject it.

License

References

[1] Afsoon Afzal, Jeremy Lacomis, Claire Le Goues, and Christopher S Timperley. A turing test for genetic improvement. In *Proceedings of the 4th International Workshop on Genetic Improvement Workshop*, pages 17–18, 2018.

[2] Md Imran Alam, Raju Halder, and Jorge Sousa Pinto. A deductive reasoning approach for database applications using verification conditions. *Journal of Systems and Software*, page 110903, 2021. doi: 10.1016/j.jss.2020.110903. URL http://dx.doi.org/10.1016/j.jss.2020.110903.

[3] Nadia Alshahwan and Mark Harman. Augmenting test suites effectiveness by increasing output diversity. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1345–1348. IEEE, 2012. doi: 10.1109/icse.2012.6227083. URL http://dx.doi.org/10.1109/icse.2012.6227083.

[4] Paul Ammann and Jeff Offutt. *Introduction to software testing.* Cambridge University Press, 2016.

[5] Saswat Anand, Corina S Păsăreanu, and Willem Visser. Jpf–se: A symbolic execution extension to java pathfinder. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 134–138. Springer, 2007.

[6] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. Automatic test generation for coverage analysis using cbmc. In *International Conference on Computer Aided Systems Theory*, pages 287–294. Springer, 2009. doi: 10.1007/978-3-642-04772-5_38. URL http://dx.doi.org/10.1007/978-3-642-04772-5_38.

[7] Andrea Arcuri. Test suite generation with the many independent objective (mio) algorithm. *Information and Software Technology*, 104:195–206, 2018. doi: 10.1016/j.infsof.2018.05.003. URL http://dx.doi.org/10.1016/j.infsof.2018.05.003.

[8] Brad Arkin, Scott Stender, and Gary McGraw. Software penetration testing. *IEEE Security & Privacy*, 3(1):84–87, 2005.

[9] Vinay Arora, Rajesh Bhatia, and Maninder Singh. A systematic review of approaches for testing concurrent programs. *Concurrency and Computation:*

Practice and Experience, 28(5):1572–1611, 2016. doi: 10.1002/cpe.3711. URL `http://dx.doi.org/10.1002/cpe.3711`.

[10] Thomas Ball and James R Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994. doi: 10.1145/143165.143180. URL `http://dx.doi.org/10.1145/143165.143180`.

[11] Earl T Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 257–269, 2015.

[12] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[13] Marcel Böhme. Stads: Software testing as species discovery. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(2):1–52, 2018.

[14] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978428. URL `http://doi.acm.org/10.1145/2976749.2978428`.

[15] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344. ACM, 2017.

[16] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[17] Bobby R Bruce, Justyna Petke, and Mark Harman. Reducing energy consumption using genetic improvement. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1327–1334, 2015. doi: 10.1145/2739480.2754752. URL `http://dx.doi.org/10.1145/2739480.2754752`.

[18] Dan Bruce, Héctor D Menéndez, and David Clark. Dorylus: An ant colony based tool for automated test case generation. In *International Symposium on Search Based Software Engineering*, pages 171–180. Springer, 2019. doi: 10.1007/978-3-030-27455-9_13. URL `http://dx.doi.org/10.1007/978-3-030-27455-9_13`.

[19] Dan Bruce, Héctor D Menéndez, Earl T Barr, and David Clark. Ant colony optimization for object-oriented unit test generation. In *International Conference on Swarm Intelligence*, pages 29–41. Springer, 2020. doi: 10.1007/978-3-030-60376-2_3. URL `http://dx.doi.org/10.1007/978-3-030-60376-2_3`.

[20] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[21] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[22] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):1–38, 2008.

[23] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. On parallel scalable uniform sat witness generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 304–319. Springer, 2015. doi: 10.1007/978-3-662-46681-0_25. URL `http://dx.doi.org/10.1007/978-3-662-46681-0_25`.

[24] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24. URL `http://dx.doi.org/10.1007/978-3-540-78800-3_24`.

[25] Kalyanmoy Deb, Ashish Anand, and Dhiraj Joshi. A computationally efficient evolutionary algorithm for real-parameter optimization. *Evolutionary computation*, 10(4):371–395, 2002. doi: 10.1162/106365602760972767. URL `http://dx.doi.org/10.1162/106365602760972767`.

[26] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002. doi: 10.1109/4235.996017. URL `http://dx.doi.org/10.1109/4235.996017`.

[27] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[28] Ibrahim K El-Far and James A Whittaker. Model-based software testing. *Encyclopedia of Software Engineering*, 2002.

[29] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[30] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012.

[31] Gordon Fraser and Andrea Arcuri. Evosuite at the sbst 2016 tool competition. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, pages 33–36, 2016.

[32] Zhoulai Fu and Zhendong Su. Achieving high coverage for floating-point code via unconstrained programming. *ACM SIGPLAN Notices*, 52(6):306–319, 2017. doi: 10.1145/3062341.3062383. URL `http://dx.doi.org/10.1145/3062341.3062383`.

[33] Jerry Gao, Xiaoying Bai, and Wei-Tek Tsai. Cloud testing-issues, challenges, needs and practice. *Software Engineering: An International Journal*, 1(1):9–23, 2011.

[34] Ali Ghanbari. Toward practical automatic program repair. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1262–1264. IEEE, 2019. doi: 10.1109/ase.2019.00156. URL `http://dx.doi.org/10.1109/ase.2019.00156`.

[35] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005. doi: 10.1007/978-3-642-19237-1_4. URL `http://dx.doi.org/10.1007/978-3-642-19237-1_4`.

[36] Arnaud Gotlieb and Matthieu Petit. A uniform random test data generator for path testing. *Journal of Systems and Software*, 83(12):2618–2626, 2010. doi: 10.1016/j.jss.2010.08.021. URL `http://dx.doi.org/10.1016/j.jss.2010.08.021`.

[37] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. *ACM SIGSOFT Software Engineering Notes*, 23(2):53–62, 1998. doi: 10.1145/271771.271790. URL `http://dx.doi.org/10.1145/271771.271790`.

[38] John J Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on systems, man, and cybernetics*, 16(1):122–128, 1986.

[39] Tao Guo, Puhan Zhang, Xin Wang, and Qiang Wei. Gramfuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation. In *2013 Second International Conference on Informatics & Applications (ICIA)*, pages 212–215. IEEE, 2013. doi: 10.1109/icoia.2013.6650258. URL `http://dx.doi.org/10.1109/icoia.2013.6650258`.

[40] Neelam Gupta, Aditya P Mathur, and Mary Lou Soffa. Generating test data for branch coverage. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, pages 219–227. IEEE, 2000.

[41] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[42] Aki Helin. Radamsa fuzzer, 2006.

[43] S Hocevar. zzuf—multi-purpose fuzzer, 2011.

[44] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. Testing concurrent programs to achieve high synchronization coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 210–220, 2012. doi: 10.1145/2338965.2336779. URL `http://dx.doi.org/10.1145/2338965.2336779`.

[45] Paul C Jorgensen and Carl Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9):30–38, 1994. doi: 10.1201/9781439889503-29. URL `http://dx.doi.org/10.1201/9781439889503-29`.

[46] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[47] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2018.

[48] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.

[49] William B Langdon and Mark Harman. Grow and graft a better cuda pknotsrg for rna pseudoknot free energy calculation. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 805–810, 2015. doi: 10.1145/2739482.2768418. URL `http://dx.doi.org/10.1145/2739482.2768418`.

[50] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604, 2017.

[51] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, 23 (5):3007–3033, 2018. doi: 10.1145/3180155.3182536. URL `http://dx.doi.org/10.1145/3180155.3182536`.

[52] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011. doi: 10.1109/tse.2011.104. URL `http://dx.doi.org/10.1109/tse.2011.104`.

[53] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[54] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485. ACM, 2018. doi: 10.1145/3238147.3238176. URL `http://dx.doi.org/10.1145/3238147.3238176`.

[55] Hareton KN Leung and Lee White. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance-1989*, pages 60–69. IEEE, 1989. doi: 10.1109/icsm.1989.65194. URL `http://dx.doi.org/10.1109/icsm.1989.65194`.

[56] Yi Li, Shaohua Wang, and Tien N Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 602–614, 2020.

[57] Jason D Lohn, William F Kraus, and Gary L Haith. Comparing a coevolutionary genetic algorithm for multiobjective optimization. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, volume 2, pages 1157–1162. IEEE, 2002. doi: 10.1109/cec.2002.1004406. URL `http://dx.doi.org/10.1109/cec.2002.1004406`.

[58] Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. Jd art: a dynamic symbolic analysis framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 442–459. Springer, 2016.

[59] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. Test generation and test prioritization for simulink models with dynamic behavior. *IEEE Transactions on Software Engineering*, 45(9):919–944, 2018. doi: 10.1109/tse.2018.2811489. URL `http://dx.doi.org/10.1109/tse.2018.2811489`.

[60] Patrick McAfee, Mohamed Wiem Mkaouer, and Daniel E Krutz. Cate: Concolic android testing using java pathfinder for android applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 213–214. IEEE, 2017. doi: 10.1109/mobilesoft.2017.35. URL `http://dx.doi.org/10.1109/mobilesoft.2017.35`.

[61] HD Menendez, M Boreale, D Gorla, and D Clark. Output sampling for output diversity in automatic unit test generation. *IEEE Transactions on Software Engineering*, 2020. doi: 10.1109/tse.2020.2987377. URL `http://dx.doi.org/10.1109/tse.2020.2987377`.

[62] László Szekeres Jonathan Metzman, Abhishek Arya, and L Szekeres. Fuzzbench: Fuzzer benchmarking as a service. *Google Security Blog*, 2020.

[63] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

[64] Ben Nagy. Crashwalk. `https://github.com/bnagy/crashwalk/`, 2015. [Online; accessed 17-February-2021].

[65] Clementine Nebut, Franck Fleurey, Yves Le Traon, and J-M Jezequel. Automatic test generation: A use case driven approach. *IEEE Transactions on Software Engineering*, 32(3):140–155, 2006. doi: 10.1109/tse.2006.22. URL `http://dx.doi.org/10.1109/tse.2006.22`.

[66] Una-May O'Reilly, Jamal Toutouh, Marcos Pertierra, Daniel Prado Sanchez, Dennis Garcia, Anthony Erb Luogo, Jonathan Kelly, and Erik Hemberg. Adversarial genetic programming for cyber security: A rising application domain where gp matters. *Genetic Programming and Evolvable Machines*, 21(1):219–250, 2020. doi: 10.1007/s10710-020-09389-y. URL `http://dx.doi.org/10.1007/s10710-020-09389-y`.

[67] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*,

pages 1–10. IEEE, 2015. doi: 10.1109/icst.2015.7102604. URL http://dx.doi.org/10.1109/icst.2015.7102604.

[68] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2017. doi: 10.1109/tse.2017.2663435. URL http://dx.doi.org/10.1109/tse.2017.2663435.

[69] Justyna Petke, Saemundur O Haraldsson, Mark Harman, William B Langdon, David R White, and John R Woodward. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 22(3): 415–432, 2017.

[70] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering*, 25(6):5193–5254, 2020.

[71] Raul Santelices, James A Jones, Yanbing Yu, and Mary Jean Harrold. Lightweight fault-localization using multiple coverage types. In *2009 IEEE 31st International Conference on Software Engineering*, pages 56–66. IEEE, 2009. doi: 10.1109/icse.2009.5070508. URL http://dx.doi.org/10.1109/icse.2009.5070508.

[72] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572, 2007.

[73] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005. doi: 10.21236/ada482657. URL http://dx.doi.org/10.21236/ada482657.

[74] K Serebryany. libfuzzer a library for coverage-guided fuzz testing. *LLVM project*, 2015.

[75] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 309–318, 2012.

[76] Kostya Serebryany. Oss-fuzz-google's continuous fuzzing service for open source software. 2017.

[77] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *ACM SIGPLAN Notices*, 37 (10):45–57, 2002. doi: 10.1145/635508.605403. URL http://dx.doi.org/10.1145/635508.605403.

[78] SN Sivanandam and SN Deepa. Genetic algorithms. In *Introduction to genetic algorithms*, pages 15–37. Springer, 2008.

[79] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 109–119, 2018. doi: 10.1145/3238147.3238172. URL http://dx.doi.org/10.1145/3238147.3238172.

[80] Dolores R. Wallace and Roger U. Fujii. Software verification and validation: an overview. *Ieee Software*, 6(3):10–17, 1989.

[81] Chao Wang, Mahmoud Said, and Aarti Gupta. Coverage guided systematic concurrency testing. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 221–230, 2011. doi: 10.1145/1985793.1985824. URL `http://dx.doi.org/10.1145/1985793.1985824`.

[82] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, pages 724–735. IEEE Press, 2019. doi: 10.1109/icse.2019.00081. URL `http://dx.doi.org/10.1109/icse.2019.00081`.

[83] Shangwen Wang, Xiaoguang Mao, and Yue Yu. An initial step towards organ transplantation based on github repository. *IEEE Access*, 6:59268–59281, 2018. doi: 10.1109/access.2018.2872669. URL `http://dx.doi.org/10.1109/access.2018.2872669`.

[84] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4):544–558, 2011.

[85] Qian Yang, J Jenny Li, and David M Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, 2009. doi: 10.1093/comjnl/bxm021. URL `http://dx.doi.org/10.1093/comjnl/bxm021`.

[86] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.

[87] M. Zalewski. American fuzzy lop, 2019. URL `http://lcamtuf.coredump.cx/afl/`.

[88] Andreas Zeller. *Why programs fail: a guide to systematic debugging.* Elsevier, 2009.

[89] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book, 2019.

[90] Jie Zhang, Muyao Zhu, Dan Hao, and Lu Zhang. An empirical study on the scalability of selective mutation testing. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 277–287. IEEE, 2014. doi: 10.1109/issre.2014.27. URL `http://dx.doi.org/10.1109/issre.2014.27`.

[91] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 2020. doi: 10.1109/tse.2019.2962027. URL `http://dx.doi.org/10.1109/tse.2019.2962027`.