AFRL-RI-RS-TR-2021-201

NEW APPROACHES FOR TEST CASE GENERATION IN SOFTWARE TESTING

WEST VIRGINIA UNIVERSITY

DECEMBER 2021

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2021-201 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WILLIAM E. MCKEEVER
Work Unit Manager

/ S /
GREGORY J. HADYNSKI
Assistant Technical Advisor
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
DECEMBER 2021	FINAL TECHNICAL REPORT	JANUARY 2020	MAY 2021

4. TITLE AND SUBTITLE

NEW APPROACHES FOR TEST CASE GENERATION IN SOFTWARE TESTING

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
FA8750-20-2-1000		62788F
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**
NOVA	WV	UI

6. AUTHOR(S)

K. Subramani

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

West Virginia University, Advanced Engineering Research Building
1220 Evansdale Drive
Morgantown, WV 26506

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505	RI	AFRL-RI-RS-TR-2021-201

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The contents of this report deal with automatic test case generation. An overview of the current field in automatic test case generation is given. A tool that combines fuzz testing, a popular test generation method, and game theory is presented. A thorough overview of combinatorial interaction testing through a structure called covering arrays is presented. Open problems with respect to covering arrays are given, as well as the impact and advantages covering arrays have in the software testing field.

15. SUBJECT TERMS

Automated Software Testing; fuzz testing and game theory; combinatorial interaction testing.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
U	U	U	SAR	46

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
WILLIAM E. MCKEEVER	**N/A**

Contents

List of Figures

List of Tables

Preface

The contents of this report deal with automatic test case generation. An overview of the current field in automatic test case generation is given. A tool that combines fuzz testing, a popular test generation method, and game theory is presented. A thorough overview of combinatorial interaction testing through a structure called covering arrays is presented. Open problems with respect to covering arrays are given, as well as the impact and advantages covering arrays have in the software testing field.

Acknowledgements

This research has been wholly supported by the Air-Force Research Laboratory, Rome through Contract FA8750-20-2-1000.

1. Summary

Software is critical to everyday life, from entertainment to national security. Software is a key element in all advanced systems and a driver of system capability, performance, security, complexity, and development risk. Therefore, it is vital that software performs as intended and is free from faults. Software testing is an integral part of the software development cycle. Software testing is the activity that attempts to verify that a program provides expected behaviors. In addition to the challenge due to size and complexity, software testing is expensive and time-consuming. It is estimated that software testing consumes 30-50% of the software development life cycle [1]. However, the alternative of deploying untested software is not a realistic approach to saving time and cost in the development process. Techniques to automate the generation of software test cases have the potential to increase the quality of tests while also reducing both cost and time to market. Increasing the effectiveness of software testing and automating the process has been a significant field of research for decades. Software testing is divided into different categories that focus on addressing various stages of the development cycle. Automatic test case generation is the process of generating input to a program to search for undesirable program behavior. There are many different approaches to generating test cases: fuzzing techniques, genetic algorithms, search-based, combinatorial, symbolic execution, and dynamic execution. A test-data generator typically takes a program or a model of the program as input and generates test data.

Many algorithms from different fields have been applied to automatic test-case generation, such as game theory, fuzz testing, or control flow graph based methods to maximize code coverage. We introduce a game theoretic test case generation algorithm based on game theory that aims to maximize path coverage by rewarding generation of test cases which cover less likely paths in the control flow graph of the program. In this algorithm, test case generation is modeled as a simultaneous partisan game with the program Control Flow Graph edges weighted with uniform probability. The selection of uniform probability distribution is based on the principle of indifference. Indifference guarantees that no prior knowledge is given regarding the paths in a random testing scheme. It is shown that the proposed fuzzer has superior performance in generating test cases that catch bugs in comparison to a random test case generator as well as other competing algorithms when applied to a dataset of programs with vulnerabilities.

We give a thorough overview of combinatorial interaction testing, which use covering arrays to generate small test suites. Covering arrays are useful to small unit tests, or minimizing test suites for software testing. Combinatorial interaction testing can also be used for hardware testing. The application of covering arrays to reduce binary test suites is well studied, but the complexity class of generating covering arrays for parameters with more potential values is unknown. Likewise, testing software can include conditions for test suites, such as required tests and constraints. Combinatorial interaction testing is capable of including these conditions, but the resulting complexity is still unknown.

2. Introduction

Software testing is a major activity in the software development cycle. The complexity and the optimality of a test vary depending on the type of the test. The common goal of all tests is to create test cases that provide evidence that the software functions correctly.

Let $\mathscr{Q}$ be a program that accepts a set of inputs $X = \langle x_1, x_2, \ldots, x_i \rangle$. A simplification is to model a program as having a mapping of a set of inputs to a set of outputs. In practice, there may be many such sub-programs contained within a single $\mathscr{Q}$. The program $\mathscr{Q}$ may also be user-input driven and cyclic, but the input/output model still applies. User interaction can be modeled as input and program response as output. $\mathscr{Q}$ can be modeled as a control flow graph G.

***Definition* 2 .0.1. (Control Flow Graph)** *A control flow graph is a directed graph* $G = (N, E, s, e)$ *where,*

1. *N is the set containing all nodes in the control flow graph,*
2. *$E = \{(n,m) \mid n,m \in N\}$ is the set of all edges that connect the nodes beginning at n and pointing to m,*
3. *s is the entry node of the control flow graph, and*
4. *e is the exit node of the control flow graph.*

The set of paths $P = \langle n_1, n_2, \ldots, n_i \rangle$ on a control flow graph consists of all possible paths through G where $(n_k, n_{k+1}) \in E$ and $n_1 = s$. A path is considered to be feasible if there exists an input $x \in X$ that traverses that path, otherwise it is infeasible.

To achieve correct functionality, two general approaches have been proposed [2]. The first being the *path-oriented* approach, which identifies control flow paths through the program and attempts to generate input test data to satisfy a certain path. The second being the *goal-oriented* approach, which identifies and executes reachable statements.

***Definition* 2.0.2. (Path-Oriented Test Data Generation)** [3] *Given a program $\mathscr{Q}$ and an execution path $p \in P$, the goal of the path-oriented test data generation problem is to find an input $x \in X$ of $\mathscr{Q}$ such that p will be traversed.*

The problem of finding all infeasible paths in a program is undecidable [4]. A problem of path-oriented test data generation is that there may be many infeasible paths that waste testing resources. In such cases, the test case generation problem can be reformulated into solving for a goal-oriented approach [2].

***Definition* 2.0.3. (Goal-Oriented Test Data Generation)** [5] *Given a program $\mathscr{Q}$ and a point in that program a node $n \in N$ for the control flow graph G, a solution to the goal-oriented test data generation problem consists of finding an input $x \in X$ of $\mathscr{Q}$ such that n is executed.*

Goal-oriented test data generation eliminates the need to select a path. Therefore, no time is wasted trying to generate input for infeasible paths. However, faults can arise from certain paths to a specific node in the control flow graph instead of the node itself. An example would be race conditions present when a certain sequence of function calls is made.

Another criterion for testing is called test coverage. Test coverage quantifies the amount of the application that has been tested using some metric. Metrics include function cov-

erage, branch coverage, code coverage, etc. The focus of this survey is only on test data generation to execute a specific path or statement in the system.

Fuzz testing (fuzzing), is an automatic test case generation method that involves inputing random test cases in an attempt to reveal faults. Fuzzing was first proposed in [6]. There are many different methods of guiding the test data that is generated by fuzzing, such as symbolic execution, search-based techniques, machine learning, constraint solving, coverage-guided, adaptive random testing, etc. Despite the diversity and expansion of fuzzing techniques being explored, the classic, basic method of generating random data has been shown to still be effective in finding faults [7]. A formal definition for fuzzing is as follows:

***Definition* 2.0.4. (Fuzz Testing)** [8] *Fuzz testing is the execution of a system using input(s) sampled from an input space to test if the system violates a correctness policy.*

The most natural way to classify different fuzzers is based on the fuzzer's awareness of the program structure. There are three categories: white-box, grey-box, and black-box. A fuzzer is classified as a white-box fuzzer if it uses source-code analysis to assist in generating test cases. A fuzzer is classified as a grey-box fuzzer if it uses instrumentation to assist in generating test cases. A fuzzer is classified as a black-box fuzzer if it does not use any information from the program being tested to generate test cases. Another way of categorizing fuzzers is whether it mutates previous test cases or generates new ones. Yet another way to categorize fuzzes is whether it is aware of the input structure of the program. Input-aware fuzzers are aware of the input structure for which data is generated whereas fuzzers that are not input-aware work by mutating a seed file to create inputs. A more in-depth description of the mechanics of fuzzing can be found in [9].

Game theory has been applied to numerous topics in the field of automated testing. Only those concerning generating test cases for goal or path oriented problems are consid-

ered. In [10], game theory is used to model finite state machines (FSM) as a game between the tester and the system under test. A strategy for the tester takes the input-output history of an FSM and determines whether a new input is selected or the test terminates. The goal of the game is to generate inputs and transitions to reach a certain state or derive conditions on the FSM. Similarly, in [11], the authors apply game theory to directed graphs for test case generation. The game explores a graph of states where vertices can be deterministic (states or nondeterministic (choice points) and edges represent transitions and have costs and probabilities. Most recently, [12] applied game theory to software testing as a game between the tester and the system under test. The authors use suspension automata (SA) modeled from software specification to create a game arena used for the optimization of test case generation. They propose a fundamental connection between specification and game arenas, test cases and game strategies, and test case derivation and strategy synthesis. They correlate a strategy in the game arena directly to test cases. The goal of the game is to reach a certain state of the SA. There have been no studies on applying game theoretic techniques directly to control flow graphs (CFGs) for test case generation.

Definition* 2.0.5. *(Two-player simultaneous partisan game [13]) *A two-player game G is defined by a pair of matrices where each element of the matrices corresponds to the payoff for each player. Each row represents the pure strategy selected by player one, and each column the pure strategy selected by player two. The goal of each player is to find a strategy that maximizes their payoff. A game is considered zero-sum if the payoff of one player is the opposite, or negation, of the payoff of the other player. Simultaneous games are defined as both players selecting their strategy at the same time. A partisan game means that each player has a different set of moves they can take that is not available to the other player.*

In table 2.1, player two picks a number between 1 and 3 while player one chooses if

		Player 2		
		1	2	3
Player 1	*Odd*	1	-1	1
	Even	-1	1	-1

Table 2.1: Example of a zero-sum two-player simultaneous partisan game

the number player two picks is going to be odd or even. Both players reveal their answers simultaneously. The table is shown from player one's perspective, but since the game is zero-sum, player two's payoff is the opposite (or negation) of the payoff matrix shown. The game is clearly partisan, as the players' choices of moves are different. For example, if player one chooses *odd* and player two selects the number *3*, then player one's payoff is 1 and player two's payoff is -1.

Most software failures are the result of one or two parameters. A study by NIST shows that almost all software failures are the result of no more than six parameters. Combinatorial testing is a fast and efficient way of creating small test suites for combinations of parameters. Combinatorial interaction testing has also been shown to have practical applications for testing artificial intelligence and machine learning. Typically, machine learning software has a high number of input parameters and values, which makes combinatorial testing ideal. Combinatorial interaction testing can also supplement other testing methods by being used to measure the coverage of their generated test suites.

Much of the research in covering arrays focuses mainly on finding greedy or meta-heuristic algorithms for constructing near-optimal covering arrays [14, 15, 16]. Greedy algorithms sacrifice optimality for speed of construction while meta-heuristic algorithms sacrifice speed for near-optimality. Greedy algorithms have found more use for practical testing, whereas meta-heuristic approaches are used in research to find new theoretical upper-bounds.

***Definition* 2.0.6. (Covering Array)** *A covering array $CA(N;t,k,v)$ of size N, strength t,*

degree k, and order v is an $N \times k$ array of v symbols in which every sub-array of t distinct columns contains every t-wise tuple of v symbols in at least one row.

0	0	0	0
0	1	1	1
1	1	0	1
1	1	1	0
1	0	1	1

Table 2.2: A Covering Array $\mathrm{CA}(5;2,4,2)$

Below is an example of 3 Boolean variables tested exhaustively versus the equivalent CA(N;2,3,2):

Test	b_0	b_1	b_2
1	0	0	0
2	0	0	1
3	0	1	0
4	0	1	1
5	1	0	0
6	1	0	1
7	1	1	0
8	1	1	1

Table 2.3: An Exhaustive Approach

Test	b_0	b_1	b_2
1	0	1	0
2	0	0	1
3	1	0	0
4	1	1	1

Table 2.4: A Covering Array Approach

The covering array above tests all possible pairwise combinations of the input parameters, reducing the total inputs to half compared to the exhaustive approach.

3. Methods, Assumptions, and Procedures

Random test case generation does not provide any assurances in terms of the coverage of all the paths in a program. Paths that are only executed by a small subset of the input domain are unlikely to be covered by random test case generation. Paths that are unlikely to be tested by randomly generated inputs could contain errors in the software. We design a complementary algorithm that tests the paths that are unlikely to be tested via random test case generation.

To model path probabilities in the CFG, the probability of an output i associated with a node j is denoted as P_{ji}. P_{ji} is calculated based on the *Principle of Indifference*, which assumes that if there are k identical outputs from a node in the CFG, each output has an occurrence probability of $\frac{1}{k}$ [17].

CFG Slicing provides a transformation tool that allows the user to focus on smaller parts of the program [18]. We use slicing to reduce the control flow graph to smaller subsets. We focus on generating inputs that provide coverage for those slices. Our method, inspired by Probabilistic Control Flow Graph Slicing (PCFG) [19], slices the CFG every time a branch is reached with equal probability among all available paths. Our slicing algorithm generates all unique slices of a program. The algorithm starts at a node with multiple output paths and creates a new unique slice by selecting only one output path and removing all other paths and nodes.

Figure 3.1 shows the CFG of Listing 3.1. The short example program is intended to

Result: Probabilistic Slicing of Control Flow Graph
Initialization
while *CFG is reducible to unique slices* **do**
    Generate CFG slice by taking one of the output paths and removing other paths and their dependent nodes
    if *Slice is non-reducible & unique* **then**
        Calculate probability of the CFG slice P by multiplying the edge probabilities p_i
        Assign utility function $-\log_2(P)$ to the slice
        Add slice to the slice set in the order of adjacency
    end
end

Algorithm 1: Probabilistic Slicing of CFG

simulate six user generated integer inputs between 1 to 10.

Listing 3.1: Motivating Example Program For Slicing

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int Phi(int a,int b=0)
{
        return (a-b);
}

int main(int argc, char** argv)
{
    srand(time(NULL));
    int x[3];
    int y[3];
```

```
    x[0]=rand()%10 + 1;   //x1 input
    x[1]=rand()%10 + 1;   //x2 input
    x[2]=rand()%10 + 1;   //x3 input
    y[0]=rand()%10 + 1;   //y1 input
    y[1]=rand()%10 + 1;   //y2 input
    y[2]=rand()%10 + 1;   //y3 input
    while(true)
    {
          x[1] = Phi(x[0],x[2]);
          y[1] = Phi(y[0],y[2]);
          if (x[1] < 10)
                  break;
          y[2]=y[1]+x[1];
          x[2]=x[1]+1;
          if(x[2] == 0)
                  break;
          }
          cout<<y[1]<<endl;
          return 0;
}
```

We model our path-oriented approach as a strategic game where the goal of the tester is to maximize the payoff by generating inputs to execute the least likely paths in the program. The game is designed by adopting a payoff for the tester which is inversely proportional to the probability of covering a path. In this case, a payoff function of $-\log_2(P_i)$ is given to the tester for testing each of the slices of the program. P_i is the probability of execution of slice

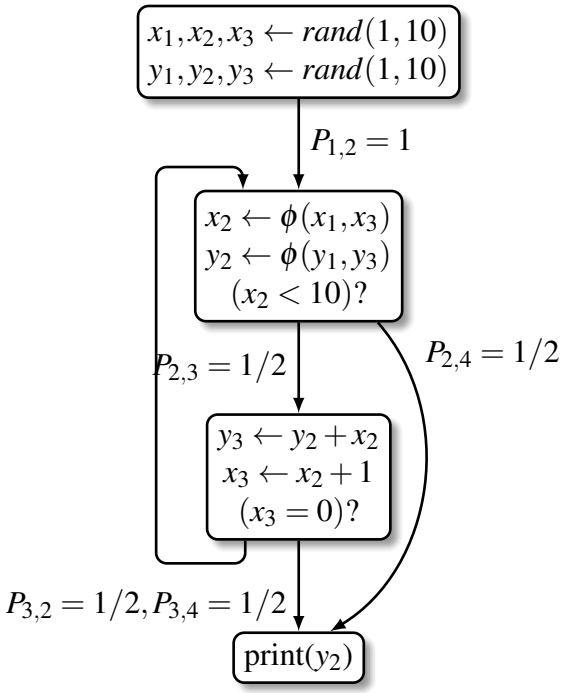

Figure 3.1: Probabilistic CFG of listing 3.1

i of the program computed as described in Algorithm 1. The payoff function ensures that the tester is motivated to generate test cases that are more likely to traverse less probable paths in the program.

In this context, the two-player game is defined as a finite strategy set $A_i \rightarrow (a_i \in A_i)$, where a_i is an action in a set of actions A_i corresponding to test cases that execute slice i of the program. The action a_i is associated with the payoff function $-\log_2(P_i)$ for the tester. The game is designed as a zero-sum game where every test case generated by the tester corresponds to a slice of the program being executed. The program is assumed to be actively playing against the tester by assigning the negation of the payoff function to the program. Table 3.1 shows the payoff matrix for the CFG in Figure 3.1. The tester has three sets of strategies A_i members of which execute the corresponding slice in the program, shown as i, with the corresponding test case as T_i. The test cases corresponding

to non-related slices are given 0 payoff to both the tester and the program.

		Program		
		b	c	d
Tester	T_b	$-\log_2(0.25)$	0	0
	T_c	0	$-\log_2(0.5)$	0
	T_d	0	0	$-\log_2(0.25)$

Table 3.1: Payoff table for the CFG slicing of motivating example program in figure 3.1

Using the best response method [20], it is evident that in Table 3.1 there are three Pure-Strategy Nash Equilibrium (PSNE) which correspond to our desired test cases. In addition, the lowest probability path that corresponds to the highest paying outcomes for the tester is considered pareto-optimal. Pareto-optimality, applied to game theory, is the notion that any other strategy would leave at least one player worse off than the current strategy.

A Mixed-Strategy Nash Equilibrium (MSNE) can also be found by balancing the utility function for the program. A MSNE for the test cases is calculated by balancing expected payoffs to the program after removing the weakly dominated rows as follows $\forall\ (i,j) \in S$:

$$P(T_i) \times \log_2(P_i) = P(T_j) \times \log_2(P_j), \tag{3.1}$$

where S is the set of all probabilistic slices in the program, $P(T_i)$ is the probability of test case number i, with $\sum_i P(T_i) = 1, 0 \leq P(T_i) \leq 1$. Finally the expected payoff of the tester in the game is given by $E(U) = -\sum_k P(T_k)^2 \log_2(P_k)$.

The MSNE probability of generating a test case for a slice is higher the lower the probability of the slice being executed. The generation strategy will effectively guarantee that the fuzzer is more likely to attempt generating test cases for lower probability slices. After each round of successful fuzzing, the slice corresponding to the test case that was executed is eliminated from the game. The payoff table is updated for the remaining slices. Inputs that execute the same slice will no longer be kept if they are generated.

Result: Game-theoretic Fuzzing
Initialization
while *Untested slices remain* **do**
    Fuzz the random seed to generate a successful fuzz with energy proportional to the payoff for the target slice
    if *Successful fuzz* **then**
        Add successful fuzz to test cases
        Remove the executed slice from the slice set
        Update payoff table
        Pick the adjacent slices
        Update the random seed with the successful fuzz
    end
end

Algorithm 2: Game-theoretic Fuzzing

Note that the MSNE probability of the program and the tester are equal in the proposed scheme. While the proposed scheme has set preferences in a PSNE, the fuzzer will always attempt to select the least probable slice due to the pareto-optimality of the slice with the lowest probability.

Efficiency is a major drawback with white-box fuzzers due to the symbolic or concolic execution involved in effectively generating test cases that traverse new paths. However, white-box fuzzers are more effective than black-box fuzzers which have no inside information of the program [21]. To achieve a balance between the efficiency and efficacy of the two techniques, grey-box fuzzing approaches use lightweight instrumentation of the program. The instrumentation determines unique identifiers for paths that are exercised by inputs and help generate new inputs based on mutating the previous successful ones.

Instead of injecting purely random inputs at the fuzz target, coverage-guided fuzzers instrument the fuzz target to collect code coverage. The fuzzer then uses this coverage information as feedback to mutate existing inputs into new ones. The fuzzer attempts to maximize code coverage by referencing all successful inputs. Two popular coverage-guided fuzzers are libFuzzer [22] and AFL [23].

Using the instrumentation, coverage-guided fuzzers monitor program execution and index paths that are taken on execution of an input. Inputs with higher coverage are prioritized. While instrumenting every basic block ensures full visibility, it reduces the efficiency of the fuzzer and thus the speed of testing.

We use the probabilistic slices created in Algorithm 1 as instrumentation guides for the fuzzer. We intend to guide the coverage to the fuzzer by indexing those inputs that execute certain slices of the program. The inputs corresponding to the successful execution of slices are then used to generate test cases for neighboring slices. Specifically, we extend AFL to implement the proposed algorithm by modifying its CFG-aware instrumentation and replacing it with slice tracking. We also base our proposed algorithm on the assumption that a fuzz that exercises a slice is more likely to generate successful fuzzes for adjacent slices. We also give AFL an energy proportional to the reward as proposed in algorithm 2.

The initial slicing will create overhead for the fuzzer. It is shown by authors in [19] that the algorithm for creating probabilistic slices has a complexity of $O(n^3)$ with respect to the number of nodes n in its CFG. However, the overhead is manageable and only run once prior to the start of the fuzzing. To further reduce the complexity of the slicing operation, we take a similar approach to the authors in [24]. A Call Graph (CG) of the program is generated, and the Intra-procedural Control Flow Graph (ICFG) of each block is taken as the target program for fuzzing.

A modified approach of the Backward Slicing used in [25] is used as our implementation of the probabilistic slicing. After slicing the ICFG of each block in the program CG, we continue by basing our fuzzer implementation on AFL [23]. AFL is one of the most successful grey-box coverage guided fuzzers used in numerous large scale software development [26, 27]. One of the major limitations of AFL, which has been addressed in previous research such as [24, 27, 28], is the allocation of constant high energy. High en-

ergy is allocated to both high and low frequency paths in the fuzzing process, causing many fuzzes to be wasted for more frequent paths. The proposed approach addresses this issue by scaling the energy to the weight of the reward given to the tester for each successful test case for a slice. The reward is taken to be inversely proportional to the probability of the slice, and the energy applied is the power schedule function $PS(i)$ for slice i given by:

$$PS(i) = \alpha * 2^{Reward(i)}, \tag{3.2}$$

where α is the energy constant used in AFL, and the reward function is:

$$Reward(i) = \max(2^{-\log_2(P_i)}, EM) \tag{3.3}$$

with P_i being the probability associated with slice i as calculated in Algorithm 1 and EM being the maximum energy available to the fuzzer for each slice as prescribed by the user. The probabilistic slicing of the ICFGs combined with the game-theoretic rewarding in the fuzzer effectively shapes the proposed solution as a directed fuzzer. The fuzzer directs test cases toward lower probability slices, as the game-theoretic model gives higher rewards for the generation of test cases for such slices.

4. Results and Discussions

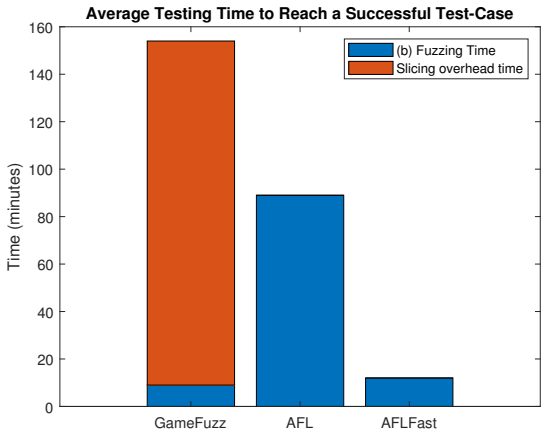

Figure 4.1: Average Testing Time to Reach a Successful Test-Case

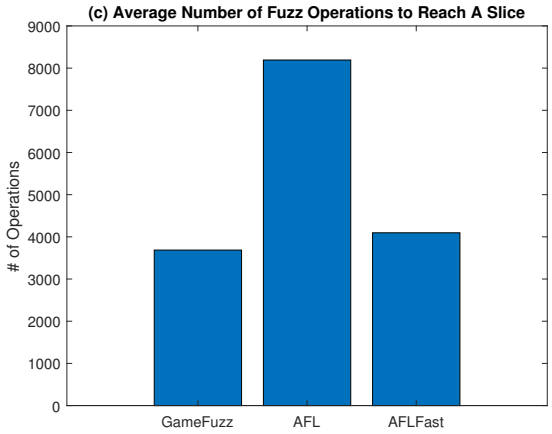

Figure 4.2: Average Number of Fuzz Operations to Reach the Lowest Probability Slice

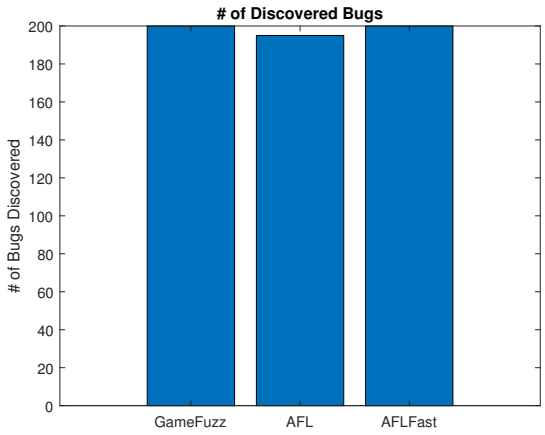

Figure 4.3: Number of Discovered Bugs

We implement our proposed improvement based on AFL 2.52b and compare our implementation with AFL and AFLFast [28]. Because our implementation is focused on slice coverage as opposed to path coverage, we can only compare the performance of the AFL and AFLFast with our implementation by comparing the number of fuzzes required to exercise paths corresponding to bugs in the slices of our approach. We utilize LAVA (Large Scale Automated Vulnerability Addition) [29] to evaluate and compare performance based on

1. General errors identified in the method,
2. Overall speed,
3. Fuzzing speed or the number of mutations required to generate successful fuzzes for exercising slices.

For practical testing, we ran our experiments on a 2.60GHz Intel Core i-7-6700HQ with four cores and Ubuntu 20.04 as the host Operating System. A maximum energy budget time of 30 minutes was allocated to each slice in the algorithm. The number of fuzzing operations required to generate a successful fuzz was tracked for each algorithm. We ran

the three fuzzers for 200 programs ranging in complexity, and the results are presented in figures 4.1, 4.2, and 4.3.

In terms of finding general errors, all methods showed similar performance. In all cases, the bugs were successfully found by all three algorithms. This is not surprising as all three methods are based on AFL. However, in some cases, it was observed that AFL ran out of time prior to fuzzing a successful test case. Running out of time is due to the 30-minute maximum time (corresponding to the maximum fuzzing energy) was not enough to generate a successful fuzz, as AFL gives equal energy to all paths alike. In terms of the overall speed AFLFast performs the best, with up to 35% less time in generating successful fuzzes that exercise the lowest probability slice compared to AFL. We did not expect GameFuzz to excel in this category due to the overhead required in slicing. However, when compared to the total number of mutations required to generate a successful fuzz for the lowest probability slice, up to 20% improvement was observed with GameFuzz compared to AFLFast.

5. Future Work

Combinatorial testing optimizes test case generation by considering input combinations. The most common type of combinatorial testing is *pairwise testing*, which uses discrete combinations of pairs of inputs to drastically reduce the number of test cases compared to an exhaustive test. Reduction of test cases is achieved by requiring all combinations of the pair of input parameters being represented at least once. Additionally, constraints can be applied to input pairs to further reduce the input space by introducing invalid combinations. Pairwise is not the only type of combinatorial testing, any strength t, where $t \geq 2$, simply requires all combinations of each *t-wise* tuple of input parameters to be represented at least once.

Authors in [30] classify algorithms for combinatorial testing as follows:

- *Algebraic*: Algebraic approaches construct the array of input parameters mathematically and solve algebraically. However, there is no general solution - the general problem of minimizing test cases that satisfies *t-wise* coverage is NP-complete [31].

- *Greedy*: Greedy approaches use a search heuristic to accumulate inputs. A greedy approach leads to sub-optimal and non-minimal results.

- *Meta-heuristic*: Meta-heuristic approaches are non-traditional algorithms that converge on near-optimal solutions but generally take longer to converge than greedy

approaches. An example of a meta-heuristic approach would be using a genetic-algorithm.

Furthermore, in the research field there is currently a focus on constraint handling for combinatorial testing algorithms warranting a section of its own.

Recently, authors in [32] have conducted an empirical comparison of combinatorial testing, random testing, and adaptive random testing. In their study, they find that combinatorial testing performs best overall. Adaptive random testing is comparable to combinatorial testing is most scenarios, however it can cost up to 3.5 times more computationally than combinatorial testing when generating highly constrained inputs. Notably, random testing performs as effectively as combinatorial and adaptive random testing with lower computational cost when a program is highly constrained but constraint information is unavailable.

5.1 Algebraic

Combinatorial testing utilizes covering arrays to reduce the test space of input parameters. In a covering array, it is sufficient to represent each *t-wise* tuple at least once. When each *t-wise* tuple is represented exactly once the object is defined to be an orthogonal array and is the minimal covering array. Orthogonal arrays are of importance to combinatorial testing because they represent the smallest reduced testing input space of a given set of parameters. Construction of orthogonal arrays and covering arrays of strength three is accomplished using a difference covering array in [33].

Formal logic can be applied to combinatorial testing to create test cases [30]. Combinatorial coverage is formalized by means of logical predicates and techniques used for solving logical problems are applied to maximize combinatorial coverage.

***Definition* 5.1.1.** *Given m input variables, each ranging in its own finite domain, a test is an assignment of values to each of the m variable* $p_1 = v_1, p_2 = v_2, ..., p_m = v_m$,

***Definition* 5.1.2.** *A pair is formally expressed as a corresponding logical expression, a test predicate* $p_1 = v_1 \wedge p_2 = v_2$.

The formal logic approach presented is capable of expressing constraints as logical predicates and are effectively handled by a formal logic tool used to solve for combinatorial coverage. In addition, the formal logic approach allows for inclusion or exclusion of select tuples for further customization of the test suite.

Combinatorial algorithms are also used in conjunction with regular expressions to generate test cases [34]. Generic test scenarios are described by means of regular expressions and whose symbols represent system operations. Values are assigned to each system operation parameter, then the regular expression is expanded to generate test cases using a combinatorial algorithm.

5.2 Greedy

A greedy approach to combinatorial test generation typically falls into one of two strategies. The first strategy is one-test-at-a-time (OTAT) and the second is one-parameter-at-a-time (OPAT). OTAT strategies attempt to cover the most *t-wise* tuples for all parameters for each test generated. A test case must cover at least one *t-wise* tuple that was previously uncovered by the test suite. The first OTAT algorithm, AETG (Automatic Efficient Test Generator), was detailed in [35]. OPAT strategies start with t (the strength of testing) parameters and create an initial covering array, another parameter is then added, more combinations are generated, and the covering array is expanded. OPAT was initially introduced by the algorithm IPO (In-Parameter-Order) [36] and then generalized, expanded, and popularized

by the algorithm IPOG [37].

5.2.1 OTAT Algorithms

The basic AETG algorithm works as follows for mixed-level covering arrays and pairwise testing [35]:

1. Choose a parameter f and a value v for f such that the parameter value appears in the greatest number of uncovered pairs.

2. Let $f_1 = f$. Then choose a random order for the remaining parameters. Then, all k parameters are ordered $f_1, ..., f_k$.

3. Assume that values have been selected for parameters $f_1, ..., f_j$. For $1 \leq i \leq j$, let the selected value for f_i be called v_i. Then, choose a value v_{j+1} for f_{j+1} as follows. For each possible value v for f_j, find the number of new pairs in the set of pairs $f_{j+1} = v$ and $f_i = v_i$ for $1 \leq i \leq j$. Then, let v_{j+1} be one of the values that appeared in the greatest number of new pairs.

Using a random seed, a number of candidate test cases M are generated using the above greedy algorithm and the candidate test case that covers the most new pairs is chosen. M can be set to any number (e.g. 10, 50, 100) and the authors use 50. Increasing M past 50 does not dramatically reduce the number of test cases that need to be generated. AETG used forbidden tuples to handle constraints. The AETG algorithm was then improved by [38] in their algorithm mAETG. mAETG expanded AETG by using combinatorial techniques to store t-sets as a rank which allowed the algorithm to handle arbitrary t-way coverage and provided less ambiguity in some edge cases as well as integrating a SAT solver for checking constraints.

After AETG came TCG [39], which was designed to be more flexible and integrated into development. TCG is capable of generating inputs for mixed-level covering arrays and pairwise interaction like AETG. Rather than choosing a random order of parameters, TCG aligns parameters in non-increasing order of number of values. Then, similarly to AETG, each test case is generated one element at a time, but distinctly, TCG uses a parameter-value data pool to store pairs, tracks how many times each pair has been selected, and selects M deterministically to be equal to the number of values of the parameter with the largest set of values. The TCG algorithm was then improved by [38] in their algorithm mTCG. mTCG expands TCG by handling all ties randomly and using a series of repeated runs and keeping only the smallest covering array generated.

5.2.2 OPAT Approaches

The basic IPO algorithm works as follows for mixed-level covering array and pairwise testing [36]:

1. Choose two parameters p_1 and p_2 and create a table T of of the corresponding values v_1 and v_2 of p_1 and p_2 consisting of rows (v_1, v_2).

2. For each remaining parameter p_i, where $3 \leq i \leq n$ and n is the number of parameters, perform horizontal and if necessary vertical growth.

    - Horizontal growth is done by appending v_i to each test $(v_1, v_2, ..., v_{i-1})$ in T to create

      $(v_1, v_2, ..., v_{i-1}, v_i)$.

    - Vertical growth is done if T does not cover all pairs between p_i and $p_1, p_2, ...p_{i-1}$ by extending T, adding a new test row $(v_1, v_2, ..., v_i)$ for each uncovered pair to the table.

IPO first constructs an initial table using two parameters, then slowly builds the table horizontally one-parameter-at-a-time. Since building the table vertically means generating more test cases, avoids vertical growth to provide a minimal covering array. In the same paper, the authors propose two more algorithms for IPO test case generation, IPO_H_EC and IPO_V, that run in polynomial time and attempt to generate minimal covering arrays. The IPO algorithm is deterministic, and thus always produces the same test set for the same inputs.

IPOG (In-Parameter-Order-General) [37] generalized IPO to handle t-way interactions for mixed-level covering arrays. IPOG initially builds a t-way covering array for the first t parameters, then extends the covering array to $t+1$, and continues extending to the next parameter until all parameters are included in the t-way covering array. IPOG suffered from long execution time and large space requirements. To rectify IPOG's weaknesses, IPOG-D [40] reduces both the space requirements and execution time by using a recursive construction procedure. The cost of these reductions is generating on average 1.5 times larger covering arrays in 1/10th of the execution time with the added restriction of parameters requiring the same number of values, otherwise known as simply a covering array.

IPO' [41] also expanded the IPO framework by generalizing to arbitrary t-way interactions for mixed-level covering arrays. IPO' broadens the search space of horizontal growth to decrease the size of the generated covering arrays and decrease execution time. Where IPO greedily selects the best value to extend one row at a time and cover the most uncovered pairs as possible, IPO' greedily searches the table and selects the best row-value pair to cover the most uncovered t-tuples as possible. IPO', as a consequence, has a larger search space for the greedy choice. In order to prevent a negative performance cost, the algorithm uses dynamic programming to store and update the coverage of each row-value pair. IPO' is both faster and more optimal than IPOG. IPO' was expanded as IPO" in [41] as well,

adding a heuristic to horizontal growth rather than computing values and greedily selecting the best one. IPO' and IPO" were implemented under the name IPOG-F and IPOG-F2, respectively.

MIPOG (Modified IPOG) [42] improved on the selection criteria of IPOG for horizontal growth and vertical growth as well as removed dependency issues, i.e. the possibility of the best value for a current test changing during vertical growth. MIPOG was designed so that it could be parallelized to multi-core processors. MC-MIPOG [42] was built from MIPOG by the authors to utilized multiple cores. The authors show that extending MIPOG to multiple cores provides a substantial speed up when parameters and coverage strength increases. MC-MIPOG also has the most optimal covering arrays when compared to previous iterations of IPO algorithms. MIPOG and MC-MIPOG are capable of producing covering arrays of strength $t > 6$. However, The authors do not compare generated covering array sizes between MIPOG and IPOG, and both MIPOG and MC-MIPOG run slower than all other iterations of IPO algorithms.

IPOG-C [43] expanded IPOG to robustly handle constraints. The authors propose and include the following optimization strategies for IPO algorithms:

- *Avoid unnecessary validity checks on t-way combinations*. Once constraints need to be considered, test cases must be checked to ensure there are no forbidden tuples. An algorithm that minimizes the number of validity checks is more optimal.

- *Check relevant constraints only*. Optimize the validity check by not checking irrelevant constraints. The less constraints in each validity check is more optimal.

- *Recording the solving history*. Time can be reduced by storing previous results.

IPOG-C models constraints as a Constraint Satisfaction Problem (CSP) and uses a constraint solver to check whether a test case satisfies the given constraints. Constraints include

both those specified by the user, as well as those that can be derived from the program. The authors point out key optimizations made to the algorithm that follow the above strategies: (1) horizontal growth will only select valid values for the parameters in the test case, so it is redundant to check constraints of a test that has only undergone horizontal growth, thus constraint checking is only done during vertical growth, (2) constraints can be modeled in a constraint relation graph where constraints that have one or more common parameters are grouped in an undirected graph and if a parameter is being checked only those constraints in that group are used for the validity check, (3) the values of parameters with constraints sent to the constraint solver and the return value is stored in a look up table so that a solving call is avoided.

5.3 Meta-Heuristic

Meta-heuristic approaches apply non-traditional algorithms to converge on solutions for combinatorial test generation. Most often, meta-heuristic techniques begin with a random set of solutions, and the initial solutions are refined by an algorithmic process and test cases are selected using a fitness function. Meta-heuristic approaches produce near-optimal solutions but their limitation is often taking more time for test generation.

Particle swarm optimization (PSO) has been applied to pairwise testing in [44]. PSO iteratively improves an initial solution by modeling the initial solution as particles moving around a search-space for optimal positions. When a better position is found, the initial solution is updated. In [44], two PSO based algorithms are proposed. One alogorithm uses an OTAT-like strategy and the other uses an OPAT-like strategy. Another PSO based algorithm was proposed by [45] in the form of a discrete particle swarm optimization (DPSO) for covering array generation. DPSO adapts set-based PSO, which utilizes set and probability theories, by introducing two auxiliary strategies to enhance performance. From the

pairwise testing PSO algorithm came a PSO algorithm for *t-wise* testing. [46] propose a particle swarm test generator (PSTG) that supports testing up to a strength of 6. The authors further optimize and improve the PSTG in [47] and propose a variable strength particle swarm test generator (VS-PSTG). [15] combine PSO algorithms with local search algorithms to produce a discrete particle swarm simulated annealing based memetic algorithm (D-PSMA). Constraints are handled for Particle Swarm Optimization (PSO) of combinatrial testing by [48]. The constraints are represented as features in an input configuration for each input parameter. A test suite is generated by using multi-objective PSO to find an optimal solution given the constraints. It is the first use of a multi-objective meta-heuristic search approach to constrained combinatorial testing. In addition, the algorithm is able to be run in parallel using multi-threading.

Colony type optimization algorithms are also well studied in respect to combinatorial test generation. The first colony type optimization applied to combinatorial test generation was the ant colony optimization algorithm [49]. Ant colony optimization algorithms locate optimal solutions by recording the quality of solutions and attempting to locate better solutions in each iteration. [49] build a test using an ant colony system - a variant of ant colony optimization - and build the test suite using a OTAT strategy. As new advances in ant colony optimization are researched, they are applied to combinatorial test generation. [50] uses fuzzy logic techniques to make a self-adapting ant colony optimization algorithm and apply it to combinatorial test generation. A variant of ant colony optimization is bee colony optimization, and [51] created an artificial bee colony for variable t-way test sets (ABCVS). Bee colony optimization produces an initial solution then generates and selects new solutions by random selection in the neighborhood of the initial solution. A new set of solutions are then chosen probabilistically from all solutions found.

5.4 Problem Definition

Definition* 5.4.1.** ***(Covering Array with Forbidden Tuples) *Let $CA(N;t,k,v)$ of size N be a covering array of strength t, degree k, and order v that is an $N \times k$ array of v symbols in which every sub-array of t distinct columns contains every t-wise tuple of v symbols in at least one row. Let C be a set of forbidden tuples, certain parameter combinations that cannot be present in the covering array. A forbidden tuple cannot be present in a row of a covering or mixed covering array.*

0	1	1	1
1	1	0	1
1	1	1	0
1	0	1	1

Table 5.1: A Covering Array $\mathrm{CA}(4;2,4,2)$ with $(0,0)$ as a forbidden tuple

Definition* 5.4.2.** ***(Covering Array with Required Tuples) *Let $CA(N;t,k,v)$ of size N be a covering array of strength t, degree k, and order v that is an $N \times k$ array of v symbols in which every sub-array of t distinct columns contains every t-wise tuple of v symbols in at least one row. Let R be a set of tuples which must be included in the covering array.*

Not all complexities are known for determining the minimum size of covering arrays as well as generating the minimum, or optimal, covering array. The complexity of generating optimal $\mathrm{CA}_{2,2}$ is known to be **P**, but the complexity of generating optimal $\mathrm{CA}_{2,2}$ with constraints is unknown.

The array coverage problem can be stated simply as generating minimal covering arrays given a set of input parameters.

	Generating Covering Arrays with Required Tuples Problem
Instance:	t, k, $v = \sum_{i=1}^{k} v_i$ and a set R of required tuples.
Question:	Generate a covering array $\mathrm{CA}(N;t,k,v)$ with the minimum value of N such that all t-way interactions and required tuples in R are covered.

	Generating Covering Arrays with Contraints Problem
Instance:	t, k, $v = \sum_{i=1}^{k} v_i$ and a set C of constraints.
Question:	Generate a covering array $\mathrm{CA}(N;t,k,v)$ with the minimum value of N such that each tuple is C-satisfying and all C-satisfying t-way interactions are covered.

We plan on determining if there is an efficient algorithm that produces the optimal size binary covering array with required and forbidden tuples. The complexity of determining the optimal size of a $\mathrm{CA}_{t,v}$ is unknown [52]. We aim to show that the problem is **NP-hard** with the additional constraints that come with required and forbidden tuples. From there, we will apply approximation techniques to determine an upper-bound on the optimal solution.

Covering arrays are an excellent tool for testing a large number of parameters that do not have a lot of potential values. Any software can be reduced to a number of inputs and the number of values that input can take. Additionally, in hardware testing, many binary input combinations need to be tested. A large number of binary parameters is a prime application for covering array testing. Test suites generated via covering arrays are used in execution, so there are no false positives when a bug is detected. Additionally, covering arrays can be applied to any resource which relies on repeated use of combinations of tasks.

On a macro level, this means that software systems that need tested together can be tested more efficiently.

Covering arrays can also be used to make efficient test suites of invalid combinations of inputs. Invalid combinations should be tested to make sure there are no missed invalid conditions or vectors for software security attacks. Covering arrays in are powerful tools for generating unit tests for correct response to invalid input while also ensuring that the test suites are as small as possible.

6. Conclusions

Game theory was leveraged to create a game-theoretic coverage-guided fuzzing approach for generating test cases to exercise less likely paths in the Control Flow Graph (CFG) of a program. The proposed approach is based on a probabilistic slicing of the program CFG and creates a probability based payoff table by modeling the test-case generation as a 2-player simultaneous game between the fuzzer and the program. A successful fuzz is taken as the seed for fuzzing an input which exercises adjacent slices. The rewards in the payoff table are used for power scheduling in the fuzzer. Our preliminary tests show promising results in reducing the number of fuzzing operations for executing low probability paths up to 20% compared to AFL. The promise of game theory as applied to software testing is proven by successful application and improvement of AFL, a standard fuzzing tool. The work resulted in a tool called GameFuzz that implements the above and works as a fuzzing tool just like AFL.

Covering arrays are a fairly recent addition to the landscape of software testing. They are best used to generate small test suites for a large number of parameters that do not have a lot of potential values. Some theoretical results are unknown for covering arrays, which need to be determined. After complexity results are established, approximation techniques and kernelization can be applied to find better upper-bounds for the size of covering arrays on any given number of input parameters. Covering arrays are great at generating a reduced test suite for unit testing to test invalid combinations of variables.

Bibliography

[1] Raquel Blanco, Javier Tuya, and Belarmino Adenso-Daz. Automated test data generation using a scatter search approach. *Information and Software Technology*, 51(4):708 – 720, 2009.

[2] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. *SIGSOFT Softw. Eng. Notes*, 23(2):5362, March 1998.

[3] Bogdan Korel. Automated software test data generation. *IEEE Trans. Software Eng.*, 16(8):870–879, 1990.

[4] Robert Jasper, Mike Brennan, Keith Williamson, Bill Currier, and David Zimmerman. Test data generation and feasible path analysis. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 94, page 95107, New York, NY, USA, 1994. Association for Computing Machinery.

[5] Bogdan Korel. A dynamic approach of test data generation. In *ICSM*, pages 311–317. IEEE, 1990.

[6] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[7] B. Miller, M. Zhang, and E. Heymann. The relevance of classic fuzz testing: Have we solved this one? *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[8] V. J. M. Mans, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[9] Patrice Godefroid. Fuzzing: hack, art, and science. *Commun. ACM*, 63(2):70–76, 2020.

[10] Mihalis Yannakakis. Testing, optimizaton, and games. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *Automata, Languages and Programming*, pages 28–45, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[11] Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. Optimal strategies for testing nondeterministic systems. In *ISSTA 2004*, volume 29 of *Software Engineering Notes*, pages 55–64. ACM, January 2004.

[12] Petra van den Bos and Marielle Stoelinga. Tester versus bug: A generic framework for model-based testing via games. *Electronic Proceedings in Theoretical Computer Science*, 277:118132, Sep 2018.

[13] X. Chen and X. Deng. Settling the complexity of two-player nash equilibrium. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 261–272, 2006.

[14] L. Cai, Y. Zhang, and W. Ji. Variable strength combinatorial test data generation using enhanced bird swarm algorithm. In *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 391–398, 2018.

[15] X. Guo, X. Song, and J. Zhou. Effective discrete memetic algorithms for covering array generation. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 01, pages 298–303, 2018.

[16] O. Moroz and V. Stepashko. Estimation of computational complexity of combinatorial-genetic algorithm combi-ga. In *2019 9th International Conference on Advanced Computer Information Technologies (ACIT)*, pages 257–260, 2019.

[17] Aditya Akundi, Eric Smith, and Tzu-Liang Tseng. Information entropy applied to software based control flow graphs. *International Journal of System Assurance Engineering and Management*, 9, 07 2018.

[18] Torben Amtoft. Slicing for modem program structures: a theory for eliminating irrelevant loops. *Information Processing Letters*, 106:45–51, 04 2008.

[19] Torben Amtoft and Anindya Banerjee. A theory of slicing for probabilistic control flow graphs. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 180–196, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[20] Thomas S Ferguson. *A Course in Game Theory*. WSPC, USA, 2020.

[21] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2577–2594. USENIX Association, August 2020.

[22] libfuzzer a library for coverage-guided fuzz testing. `https://llvm.org/docs/LibFuzzer.html`.

[23] American fuzzy lop (afl). `https://lcamtuf.coredump.cx/afl/`.

[24] 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*, CCS '17, page 23292344, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. 2016.

[26] L. Situ, L. Wang, X. Li, L. Guan, W. Zhang, and P. Liu. Energy distribution matters in greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 270–271, 2019.

[27] C. Lemieux and K. Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 475–485, 2018.

[28] 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, page 10321043, New York, NY, USA, 2016. Association for Computing Machinery.

[29] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, 2016.

[30] Andrea Calvagna and Angelo Gargantini. A formal logic approach to constrained combinatorial testing. *J. Autom. Reasoning*, 45:331–358, 12 2010.

[31] Alan W. Williams and Robert L. Probert. Formulation of the Interaction Test Coverage Problem as an Integer Program. In *Proceedings of the 14th International Conference on Testing Communicating Systems: Applications to Internet Technologies and Services*, volume 210 of *IFIP Conference Proceedings*, pages 283–None. Kluwer, 2002.

[32] H. Wu, C. Nie, J. Petke, Y. Jia, and M. Harman. An empirical comparison of combinatorial testing, random testing and adaptive random testing. *IEEE Transactions on Software Engineering*, 46(3):302–320, 2020.

[33] Lijun Ji and Jianxing Yin. Constructions of new orthogonal arrays and covering arrays of strength three. *Journal of Combinatorial Theory, Series A*, 117(3):236 – 247, 2010.

[34] M. P. Usaola, F. R. Romero, R. R. Aranda, and I. G. Rodrguez. Test case generation with regular expressions and combinatorial techniques. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 189–198, 2017.

[35] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The aetg system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[36] Yu Lei and K. C. Tai. In-parameter-order: a test generation strategy for pairwise testing. In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No.98EX231)*, pages 254–261, 1998.

[37] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. Ipog: A general strategy for t-way software testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, pages 549–556, 2007.

[38] Myra B. Cohen. *Designing Test Suites for Software Interactions Testing*. PhD thesis, University of Auckland, 2004.

[39] Yu-Wen Tung and W. S. Aldiwan. Automating test case generation for the new generation mission software system. In *2000 IEEE Aerospace Conference. Proceedings (Cat. No.00TH8484)*, volume 1, pages 431–437 vol.1, 2000.

[40] Yu Lei, Raghu N. Kacker, David R. Kuhn, Vadim Okun, and James F. Lawrence. Ipog/ipog-d: Efficient test generation for multi-way combinatorial testing. *Software Testing Verification and Reliability*, 18, 2007.

[41] M. Forbes, J. Lawrence, Yu Lei, Raghu Kacker, and Richard Kuhn. Refining the in-parameter-order strategy for construction covering arrays. *Journal of Research of the National Institute of Standards and Technology*, 113(5):287–297, 2008.

[42] Mohammed I. Younis and Kamal Z. Zamli. Mc-mipog: A parallel t-way test generation strategy for multicore systems. *ETRI Journal*, 32(1):73–83, 2010.

[43] L. Yu, Y. Lei, M. Nourozborazjany, R. N. Kacker, and D. R. Kuhn. An efficient algorithm for constraint handling in combinatorial test generation. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 242–251, 2013.

[44] X. Chen, Q. Gu, J. Qi, and D. Chen. Applying particle swarm optimization to pairwise testing. In *2010 IEEE 34th Annual Computer Software and Applications Conference*, pages 107–116, 2010.

[45] H. Wu, C. Nie, F. Kuo, H. Leung, and C. J. Colbourn. A discrete particle swarm optimization for covering array generation. *IEEE Transactions on Evolutionary Computation*, 19(4):575–591, 2015.

[46] Bestoun Ahmed and Kamal Zamli. Pstg: A t-way strategy adopting particle swarm optimization. In *Proceeding of Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation*, pages 1–5, 01 2010.

[47] Bestoun S. Ahmed and Kamal Z. Zamli. A variable strength interaction test suites generation strategy using particle swarm optimization. *Journal of Systems and Software*, 84(12):2171 – 2185, 2011.

[48] Bestoun S. Ahmed, Luca M. Gambardella, Wasif Afzal, and Kamal Z. Zamli. Handling constraints in combinatorial interaction testing in the presence of multi objective particle swarm and multithreading. *Information and Software Technology*, 86:2036, Jun 2017.

[49] X. Chen, Q. Gu, A. Li, and D. Chen. Variable strength interaction testing with an ant colony system approach. In *2009 16th Asia-Pacific Software Engineering Conference*, pages 160–167, 2009.

[50] Mohd Zamri Zahir Ahmad, Rozmie Razif Othman, Mohd Shaiful Aziz Rashid Ali, and Nuraminah Ramli. A self-adapting ant colony optimization algorithm using fuzzy logic (ACOF) for combinatorial test suite generation. *IOP Conference Series: Materials Science and Engineering*, 767:012017, mar 2020.

[51] Ammar K Alazzawi, Helmi Md Rais, and Shuib Basri. Abcvs: An artificial bee colony for generating variable t-way test sets. *International Journal of Advanced Computer Science and Applications*, 10(4), 2019.

[52] Ludwig Kampel and Dimitris E. Simos. A survey on the state of the art of complexity problems for covering arrays. *Theor. Comput. Sci.*, 800:107–124, 2019.

Acronyms

FSM	Finite State Machine
SA	Suspension Automata
CFG	Control Flow Graph
NIST	National Institute of Standards and Technology
CA	Covering Array
PCFG	Probabilistic Control Flow Graph
PSNE	Pure Strategy Nash Equilibrium
MSNE	Mixed-Strategy Nash Equilibrium
AFL	American Fuzzy Lop
CG	Call Graph
ICFG	Intra-procedural Control Flow Graph
LAVA	Large-Scale Automated Vulnerability Addition
NP	Nondeterministic Polynomial
OTAT	One-Test-At-a-Time
AETG	Automatic Efficient Test Generator
IPOG	In-Parameter-Order-General
OPAT	One-Parameter-At-a-Time
TCG	Test Case Generation
IPO	In-Parameter-Order
MIPOG	Modified In-Parameter-Order-General
MC-MIPOG	Multicore-Modified Input Parameter Order
IPOG-C	In-Parameter-Order-General - Combinatorial Testing
CSP	Constraint Satisfaction Problem
PSO	Particle Swarm Optimization
DPSO	Discrete Particle Swarm Optimization
PSTG	Particle Swarm Test Generator
VS-PSTG	Variable Strength Particle Swarm Test Generator
D-PSMA	Discrete Particle Swarm Simulated Annealing Based Memetic Algorithm
ABCVS	Artificial Bee Colony for Variable t-way test Sets