SAND2022-6592C

Algorithmic Input Generation for More Effective Software Testing

Laura Epifanovskaya
Institute for Defense Analyses
Alexandria, VA, USA
lepifano@ida.org

Reginald Meeson
Institute for Defense Analyses
Alexandria, VA, USA
rmeeson@ida.org

Christopher McCormack
Institute for Defense Analyses
Alexandria, VA, USA
cmccorma@ida.org

Jinseo R. Lee
Institute for Defense Analyses
Alexandria, VA, USA
jlee@ida.org

Robert C. Armstrong
Sandia National Laboratories
Livermore, CA, USA
rob@sandia.gov

Jackson R. Mayo
Sandia National Laboratories
Livermore, CA, USA
jmayo@sandia.gov

***Abstract*—It is impossible in practice to comprehensively test even small software programs due to the vastness of the reachable state space; however, modern cyber-physical systems such as aircraft require a high degree of confidence in software safety and reliability. Here we explore methods of generating test sets to effectively and efficiently explore the state space for a module based on the Traffic Collision Avoidance System (TCAS) used on commercial aircraft. A formal model of TCAS in the model-checking language NuSMV provides an output oracle. We compare test sets generated using various methods, including covering arrays, random, and a low-complexity input paradigm applied to 28 versions of the TCAS C program containing seeded errors. Faults are triggered by tests for all 28 programs using a combination of covering arrays and random input generation. Complexity-based inputs perform more efficiently than covering arrays, and can be paired with random input generation to create efficient and effective test sets. A random forest classifier identifies variable values that can be targeted to generate tests even more efficiently in future work, by combining a machine-learned fuzzing algorithm with more complex model oracles developed in model-based systems engineering (MBSE) software.**

Keywords—covering arrays, complexity, software testing, reliability, formal models

I. Introduction

The power, capacity, and flexibility of digital technology provide capability to modern cyber-physical systems such as vehicles, aircraft, and weaponry that was simply out of reach for older analog and mechanical systems. The power of digital technology, however, comes at a price. Apart from the potential vulnerabilities created by networked systems, a computer-controlled system is also only as secure, reliable, and resilient as the software that controls it. In the past few decades, flaws and vulnerabilities in software led to devastating accidents and breaches that compromised personal information, cost billions of dollars, and in the case of software-powered medical devices, transportation, or other safety-critical systems, sometimes led to injuries or even loss of life [1].

Achieving software assurance through testing is hard. Testing based on software requirements—the behaviors the software was engineered to produce—can demonstrate the presence or absence of nominal functionality. However, test sets that achieve requirements-based coverage are not necessarily sufficient to exercise all of the software's latent behaviors [2]. One approach to address the problem is to test in a way that provides coverage of the software structure: the branching logic pathways made possible by conditional statements in the program under test. In avionics, for example, structural coverage criteria are used to determine test suite adequacy for certification purposes [3]. Previously, standard guidance used by certification authorities across the U.S., Canada, and Europe required safety-critical software testing to demonstrate structural coverage [4] [5].

Research has raised questions about the effectiveness of coverage-based testing, finding in some cases that test sets generated using coverage criteria are less effective than random tests of the same length [6]. One possible reason for this is masking, in which changing the value of a condition does not affect the outcome of the assertion [7]. Full characterization of the state space reachable by software programs is effectively impossible; that is, while cyber-physical system programs have a finite state space and therefore the problem of fully characterizing program behavior is in principle decidable, in practice it is combinatorially hard [9]. In practice, software test design is an inexact science that leverages any number of techniques to attempt to exercise as much of a program under test as possible within the schedule and budget allotted to test activities. Improving software testing is still an area of active and urgent research.

In this paper, we compare various methods of generating test suites based on their efficiency and effectiveness at triggering faults across a set of programs deliberately seeded with coding errors, a type of mutation testing. The rest of the paper is organized as follows: Section 2 provides background on the types of test methods we employ; Section 3 provides an overview of the experimental design, including a description of

the program under test; and Section 4 reports experimental results, provides analysis and context, and sets out our conclusions.

II. Background

Earlier avionics software certification guidance has since been replaced by updated guidelines that make allowance for new developments in system design and engineering, including model-based development paradigms, and contain considerations for replacing some typical software testing activities with simulation and formal methods [10]. Formal methods are techniques and tools that apply mathematical rigor to software design and verification activities. Using formal methods, software designers encode software behaviors in terms of formal logic to create "correct-by-construction" programs. This helps to eliminate software bugs in three ways: (1) the careful, logical design prevents accidental introduction of unintended behaviors; (2) the final software product is formally verifiable using tools like model checkers and theorem provers; and (3) the formal specification of a design model prior to the programming process prevents faulty specifications from introducing undesired behaviors into well-written code. For example, "race conditions", where program outputs change based on uncontrolled sequences of events, can arise in systems with inadequate controls. Model checkers can prove that a design under test does not contain specific flaws that create safety or reliability faults. When applied in the system design phase, formal methods are extremely powerful in eliminating software bugs and developing cyber-secure systems [11] [12]. These techniques, however, are of limited utility when applied to legacy software or software that has already been written and has entered the test phase.

However, some tools from formal methods can still be usefully applied to the software testing process. For example, one of the most daunting tasks of any software test effort is determining what the expected behavior, or output, of a program should be given a particular input. Correct outputs are hard to formulate a priori, but they are needed to determine whether a software test has produced a faulty output. The correct output is usually only known if there is a "golden copy" of the software program that is somehow verified to be correct, usually though extensive testing and use. Formal models can be used to produce software "oracles" that provide correct outputs when presented a set of inputs. This is done by specifying a set of requirements as properties in formal language and checking the model design against these properties; the requirements are formulated as "trap properties" deliberately written as negations of the design requirements so that the model checker will declare each negated property false and proceed to output a counterexample that provides a step-by-step proof together with the inputs that produce the falsified output. The oracle output is then checked against the program output to verify correct program behavior [13] [14] [15].

Apart from the need for a test oracle, a separate consideration in testing is deciding upon useful sets of test inputs. As noted above, requirements-based testing and coverage-based testing are not by themselves sufficient to uncover all faulty behaviors produced by computer programs. One approach to generating inputs is to focus on those that may trigger latent faulty behaviors with higher probability than randomly-generated inputs. For example, the Design of Experiments (DOE) method creates a "full factorial" test set by generating all possible combinations of variable values. The logic behind this approach maintains that faults are more likely to be discovered in tests through variable interactions [16]. Since in a typical program the range of possible variable values makes a full factorial set over all possible values infeasible, the test designer uses equivalence partitioning to allocate each variable a single allowed value per equivalence bin [17]. Even with equivalence partitioning, however, full-factorial test sets for software can be unmanageably large. A solution to this is the covering array. Covering arrays are a mathematical construct wherein a full factorial test set of variables is reduced in size while preserving a specified strength, t, of variable interactions (all combinations of t variables are covered). In this construction, the number of test input strings grows only logarithmically in the number of parameters instead of exponentially [18].

Another popular approach to software testing that does not rely on oracles is algorithm-directed fuzzing, a form of automated random testing. Algorithm-directed fuzzing is widely-used and shown to be effective. Fuzzing is responsible for uncovering the vast majority of known remote code execution and privilege-escalation bugs [19]. Companies like Google use guided fuzzing—fuzzing that leverages semantic knowledge to generate input strings—to continuously test software products. Guided fuzzing is powerful because of its ability to produce "corner case" inputs, that is, inputs semantically similar enough to expected program inputs that the software accepts them and attempts to execute them, but that contain some atypical component that can trigger unexpected program behavior [20]. Testing boundary conditions using guided fuzzing is an efficient way of locating sources of undesired behaviors. Efficiency is important in software testing because, while it may be possible in principle to test every state of a given program, the time and resources required to do so are prohibitive in practice. It is necessary to gain confidence in a program's reliability with limited resources, including time and budget. An efficient test set is one that maximizes the number of faults triggered per test input, while minimizing the required number of test inputs and the length of time required to test.

We will compare covering arrays, random testing, low-complexity test suites, and a form of guided fuzzing based on their efficiency and effectiveness at triggering faults in the error seeded programs.

III. Experiment

A. *Traffic Collision Avoidance System (TCAS)*

We performed our software tests on a small C-language module of the Traffic Collision Avoidance System (TCAS), an airborne system used in commercial aviation to reduce the risk of mid-air collisions. When the system detects another transponder-equipped aircraft within close proximity, it alerts the pilot and issues an advisory in order to avoid a crash. When

TCAS detects another transponder 20-48 seconds away from a potential collision, it sends the pilot a traffic advisory (TA). If no action is taken to prevent the collision, TCAS sends the pilot a resolution advisory (RA).

The program module under test takes an input string of 12 variables and outputs a single variable, called *alt_sep*, that is assigned the value of the RA signaled by the host aircraft. The *alt_sep* variable has three possible values: *UNRESOLVED*, *UPWARD_RA*, and *DOWNWARD_RA*.

The design of our experiment replicated work by Vadim Okun and Richard Kuhn at the U.S. National Institutes for Standards and Technology (NIST) [21]. Kuhn and Okun use covering arrays of varying interaction strengths as their test sets, and they use the model checker NuSMV to generate oracle output values for each input set by specifying trap properties, as described in Section II. We first separated the TCAS input variable values into the equivalence bins used by Kuhn and Okun so that we could directly compare our test results with theirs. Equivalence partitioning simplifies model checking in NuSMV by limiting the number of variable assignments to be checked; it also reasonably constrains the number of inputs in a covering array. A single value from each equivalence partition is assigned as the representative value from that equivalence bin. The bins values assigned to each variable are found in Table 1.

To compare test approaches according to their effectiveness and efficiency in finding bugs in the program unit under test, we generated several different test suite types. We then ran each test suite as inputs to a set of TCAS executables and monitored the output using the NuSMV model output for a given input variable combination as our oracle. We employed a form of mutation testing in which we manually introduced minor changes to the correct TCAS module to create a new, buggy program. Conditional operator values were changed in some programs (for example, from > to <), values of internal variables were manipulated in others, and arithmetic operators were changed in still others (for example, a + operator might become a -). This approach generated a suite of 28 buggy TCAS program executables containing a single mutation each.

TABLE 1. TCAS VARIABLE VALUES

TCAS Variable	Equivalence Bin Values
Cur_Vertical_Sep	299, 300, 601
High_Confidence	TRUE, FALSE
Two_of_Three_Reports_Valid	TRUE, FALSE
Own_Tracked_Alt	1, 2
Own_Tracked_Alt_Rate	600, 601
Other_Tracked_Alt	1, 2
Alt_Layer_Value	0, 1, 2, 3
Up_Separation	0, 399, 400, 499, 500, 639, 640, 739, 740, 840
Down_Separation	0, 399, 400, 499, 500, 639, 640, 739, 740, 840
Other_RAC	NO_INTENT, DO_NOT_CLIMB, DO_NOT_DESCEND
Other_Capability	TCAS_TA, OTHER
Climb_Inhibit	TRUE, FALSE

B. *Covering Arrays*

Replicating Kuhn and Okun's approach, we created covering arrays of the variables in Table 1 to use as our starting test sets. JMP Pro 15 statistical software generated a set of *t*-way covering arrays for *t*=2 to *t*=6-way interactions. Because the JMP covering array optimization algorithm does not always reach the theoretical minimum number of *t*-way inputs, covering arrays of the same strength t with the same input variables do not always contain the same number of tests. The array sizes we used in the initial covering array tests are in Table 2. The minimum test set size is limited by the number of equivalence bins in the *t* variables with the most values for each t-way test; for example, the 2-way covering array has a minimum of 10 x 10 = 100 values because the *Up_Separation* and *Down_Separation* variables have 10 possible assignments each.

C. *Random Testing*

Kuhn and Okun report 100% success rates in their mutation testing of the TCAS module using *t*=5 and *t=6*-way covering arrays. This might reflect the power of test sets generated using covering arrays; on the other hand, it might be a result of the relative sizes of the *t*=5 and *t=6*-way test sets (with 4220 and 10,902 test inputs, respectively) compared to the fewer-way interaction test sets (with 100, 405, and 1375 tests). Test size is known to correlate positively with the number of bugs found, simply because the increased variety of test input values creates additional reach in the software state space [21]. To determine whether the strength of the covering array interaction or the sheer number of test inputs is responsible for the effectiveness in triggering faults in the C executables, we generated random test sets with numbers of input values corresponding to those in the *t=2, 3,4,5, and 6*-way covering arrays.

D. *Kolmogorov Complexity*

Covering arrays provide one means of reducing the possible input space to a program in a targeted way, with a focus on preserving variable interactions. Another approach to a targeted reduction of input space is selecting the relatively small number of inputs of low information complexity. The length of the shortest description of a bit string is known as its

TABLE 2. T-WAY COVERING ARRAY TEST SETS

Array Strength	Number of Tests
2-way	100
3-way	400
4-way	1215
5-way	3607
6-way	11018

Kolmogorov complexity [23]. Based on the observation that faults arise from "corner case" inputs [20], we tested the relatively small number of inputs of low Kolmogorov complexity to determine if they are more effective at triggering faults than other inputs.

This approach was initially developed to address cybersecurity-related attacker/defender scenarios [9]. If both attacker and defender are forced to select inputs at random from a combinatorially large set of possibilities, both are equally disadvantaged. Because the vast majority of inputs are algorithmically random, a strategy to bias inputs to those more likely to trigger faults is to focus on the more characterizable space of low-complexity inputs. A password checker provides a concrete example of why this may be a good strategy for both attackers and defenders. If a correct password is treated as a deliberately seeded fault that, when triggered, provides access to a computer system, then a weak password corresponds to a low-complexity input string. Given no prior information about the password, it behooves both attacker and defender to fully test this input space to determine whether any low-complexity inputs trigger the password "fault".

To test this empirically, we generated a set of low-complexity inputs from short descriptions by "decompressing" small programs in the golfing language 05AB1E, which we chose based on the clarity of its documentation and the availability of one- and two-byte program descriptions on GitHub [24]. We used the set of all 65,536 possible two-byte programs in order to have enough inputs to compare with the covering arrays. Well-formed low-complexity TCAS inputs were generated by mapping the program output onto values from the variable equivalence bins, preserving the amount of information (or increasing it by no more than an additive constant). See Table 3 for examples.

E. *Guided Fuzzing-Based Tests*

Finally, we generated test inputs using a different complexity measure based on a kind of Hamming distance from a seed string. This approach was based on the observation that, when a bug is found in a program, there are often other faults nearby that can be triggered with similarly structured inputs [9]. This is the premise on which many modern-day fuzzing algorithms are designed; American Fuzzy Lop (AFL), for example, uses a genetic algorithm to iterate on inputs that generate faults or other "interesting" program behaviors [25]. This fuzzing algorithm, and guided fuzzing in general, have proven more effective at finding bugs than fully random automated testing [26].

To develop these test sets, we first selected test inputs that resulted in *UPWARD_RA* or *DOWNWARD_RA* outputs to use as seeds, because they are rare compared to *UNRESOLVED* outputs. The program logic in the C program that controls the assignment of the values *UPWARD_RA* and *DOWNWARD_RA* has the form:

```
if (input value >= control value) {
    alt_sep = UPWARD_RA}
else {alt_sep = DOWNWARD_RA}
```
(1)

Because of the equivalence partitioning of the input variables, the practical result is an average 1:2 ratio of DOWNWARD_RA to UPWARD_RA outputs in large test sets. *UNRESOLVED* outputs are the most numerous by far. As an example, our covering array test set of 100 inputs resulted in 95 *UNRESOLVED* outputs, five *UPWARD_RA* outputs, and no *DOWNWARD_RA* outputs. The comparative rarity of input strings that produce DOWNWARD_RA outputs makes them a kind of "golden input" that traverses uncommon paths through the C program. Using inputs selected at random from the set that produces these rare outputs, new inputs were generated through small permutations to the variable values. Applying our complexity metric, one unit of Hamming distance corresponds to a change of a single variable in an input string to a different equivalence bin value. Any new variable value (in the equivalence set) is allowed. So, using an example line of TCAS input:

299 0 0 2 600 2 0 500 499 0 1 0

We generate a new input of Hamming distance 1 by, for example, changing the value of the first variable from 299 to 300. The new input is:

300 0 0 2 600 2 0 500 499 0 1 0

Some other example inputs of Hamming distance one from the original input are:

299 ***1*** 0 2 600 2 0 500 499 0 1 0
299 0 0 2 600 2 0 500 ***740*** 0 1 0
299 0 0 ***0*** 600 2 0 500 499 0 1 0

All input strings with a Hamming distance of one were stored in a test set named *Tier 1*. *Tier 2* test sets were generated by changing two different variable values, and *Tier 3* inputs change the value of three variables in a seed. We generated five different test sets in Tier 1, Tier 2, and Tier 3 using five different seed inputs. One seed input string was chosen at random from the subset of inputs resulting in *UPWARD_RA*. The second seed was a string resulting from Tier 3 changes to that input. The third test set was seeded with a *DOWNWARD_RA*-output string (again, chosen at random from the subset of all strings that produce a *DOWNWARD_RA* output). Test sets four and five were generated using strings from Tier 3 of the original *DOWNWARD_RA* string. Overall, this approach resulted in 15 test sets: three tiers resulting from changing five different seed

TABLE 3. 05AB1E TCAS TEST INPUTS

05AB1E Program	**žN**	**žQ**	**žS**
Output	bcdfghjklmnp	!"#$%&'()*+,	qwertyuiop00
Indexed Output	1 0 0 0 0 0 1 4 4 1 1 1	0 0 0 0 0 0 0 1 1 0 0 0	1 0 0 0 0 0 1 4 4 1 0 0
TCAS Input	300 1 1 1 600 1 2 500 500 1 1 1	299 1 1 1 600 1 1 399 399 0 0 0	300 1 1 1 600 1 2 500 500 1 0 0

strings. Different Hamming distances resulted in different numbers of test inputs for each tier: Tier 1 test sets contain 32 tests, Tier 2 test sets contain 429 tests, and Tier 3 test sets contain 2988 tests.

IV. Results

A. *Covering Arrays and Random Tests*

Kuhn and Okun reported a 100% success rate using covering arrays of interaction size *t*=5 or higher. As Table 4 shows, our five- and six-way covering arrays were not as successful; they caught all but one of the seeded errors (faults were triggered in 27 out of the 28 programs tested). These results, along with those from other approaches (including ones described below), are summarized in Fig. 1.

The data from these tests raised two questions. First, why was the bug in the 28th program not triggered by any of the test inputs? Second, are we catching more bugs because of the structure of the covering arrays or because we are running many more tests at increasing t-way interaction sizes?

To answer the first question, we investigated the bug that was not triggered. The seeded error was a small change in value in one line of code that takes the input variable *Cur_Vertical_Sep* and uses it to set the value of an internal program Boolean variable. The correct TCAS program contains a conditional statement that sets an internal Boolean variable called *enabled* to 1 if *Cur_Vertical_Sep* is greater than the internal program variable *MAXALTDIFF*. The variable *MAXALTDIFF* is set to 600 in the correct TCAS program and 500 in the buggy program. The reason our test inputs were unable to trigger the error is that the equivalence bin values we chose for *Cur_Vertical_Sep* were 299, 300, and 601.

This set of values does not enable a tester to detect a change in the range 500-600, because as long as *MAXALTDIFF* has a value in the range 301-600, the conditional statement *Cur_Vertical_Sep* > *MAXALTDIFF* is true if *Cur_Vertical_Sep* = 601 and false if *Cur_Vertical_Sep* = 300. From this result it becomes clear that values in equivalence bins are not always equivalent if there is an error in the program under test.

In order to determine whether we are uncovering more errors with *t*=5 and 6-way test sets because of the power of the covering arrays or simply because the test sets are larger, we compared the covering array test results to results using the randomly-generated test sets of equal size. The randomly generated test sets performed similarly to the covering arrays, although they did catch one fewer buggy program in both *t*=5 and 6-way tests (test inputs triggered faults in only 26 of 28 buggy programs). As Table 5 shows, the random inputs did better than the covering array inputs at smaller *t*-way interaction sizes.

The covering array tests demonstrated that, for arrays with high enough *t* (*t*=5 and *t*=6), the arrays are modestly more effective than randomly selected inputs at triggering faults. Upon inspection, we found that a bug that was never caught by any of the randomly generated test sets was triggered by all of the covering array test sets except for the smallest, pairwise interaction set (*t*=2). The bug was in a location in the code only reachable with one unique combination of six of the input variables (in addition to other criteria being met). The unique combination of those six variables occurs rarely in the set of all possible combinations; therefore, the randomly generated test sets were not certain to contain the necessary combination. In other words, the bug that was not caught by the randomly generated inputs is one that is hard to catch, statistically speaking. The *t*=6-way covering array, on the other hand, is guaranteed to contain the variable combination. To illustrate why that is the case, consider a unique specification of the 6-way set of variables with the most equivalence tiers. The prevalence of that unique combination in the set of all possible values is:

$$\frac{1}{10} \times \frac{1}{10} \times \frac{1}{4} \times \frac{1}{3} \times \frac{1}{3} \times \frac{1}{2} = \frac{1}{7200} \quad (1)$$

The six-way covering array and like-sized random test set contain 11016 inputs. For the random set, the likelihood of drawing 11016 inputs that do *not* contain the desired six values is:

$$(1 - \frac{1}{7200})^{11016} \sim 0.22 \quad (2)$$

Therefore, the likelihood of drawing a set at random that contains the values is 78%, whereas the 6-way covering array is guaranteed to contain the input. This imparts some advantage to the larger *t*-way arrays in triggering faults located along low-probability traces of the code.

The smaller *t*-way arrays do not provide the same advantage. For example, given the same 6-way variable, the *t*=3-way covering array is guaranteed to contain the first 3-way combination of values, but when paired with the remaining three values, the chances of ending up with the required six values sequentially are essentially the same as in the randomly generated set. While still possible to draw the rare input combination that triggers the fault in question at random, the likelihood is decreased relative to the covering arrays that it will

Table 4. Covering Array Test Results

t (strength)	t=2	t=3	t=4	t=5	t=6
Test Size	100	400	1215	3607	11018
Bugs Caught	4	16	21	27	27
Test Failures	103	257	1292	3892	11663
Total Tests	2800	11200	34020	100996	308504
% Efficiency	3.7	2.3	3.8	3.9	3.8

Table 1. Random Test Results

t (strength)	t=2	t=3	t=4	t=5	t=6
Test Size	100	400	1215	3607	11018
Bugs Caught	4	19	23	26	26
Test Failures	78	351	1035	2957	8878
Total Tests	2800	11200	34020	100996	308504
% Efficiency	2.7	3.1	3.0	2.9	2.9

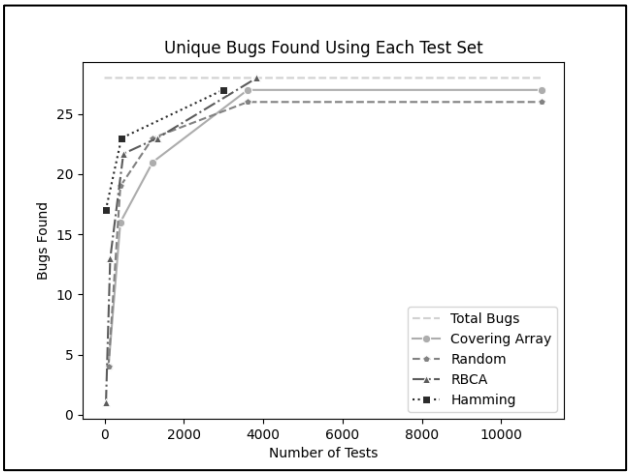

Fig. 1. Summary of Test Methods

appear in a test set at the larger test set sizes.

Most of the fault-finding effectiveness of the test suites appears to result from the size of the larger sets; the large t-way covering arrays do add some value by forcing the generation of certain input combinations that are less likely to be selected in a random draw. On the other hand, the statistical unlikelihood of finding the input in a random draw could be overcome by executing multiple random draws.

B. *Random from equivalence Bin Covering Array (RBCA)*

Because each approach has specific advantages—the covering array approach targets low-probability code branches and the random input approach covers the entire space within an equivalence bin, allowing for more granularity in value coverage—we decided to combine the approaches into one. In this new approach, we specified equivalence bins using values from the first covering array tests as upper and lower bounds for new equivalence bins; we then wrote a script that generates a random value within that equivalence bin to populate the test input. Thanks to this new bin specification method, these *t*-way interaction sizes resulted in smaller test sets than the simple covering array *t*-way interaction tests. The ten equivalence bins in the *Up_Separation* and *Down_Separation* variables, for example, were reduced to six: 0-399, 400-499, 500-639, 640-739, 740-840, 840-1000. The reduction in the number of bins for these two variables resulted in smaller covering array sizes using this method, which we named RBCA (***R***andom from equivalence ***B***in ***C***overing ***A***rray).

Once we had generated covering arrays using the levels of each input variable equivalence bin as placeholders, we ran a script to populate each placeholder with a randomly selected value within the bin. In this way, RBCA leverages both the interaction-maximizing power of the covering array and the range of the random-value generator. The inability of the first covering arrays to trigger the 28th bug is resolved by the random assignment of the *Cur_Vertical_Sep* variable to a number within the equivalence bin 500-639. Results are shown in Table 6. The RBCA method was successful in triggering faults in all eight of the buggy programs at *t*=6. The *t*=6-way RBCA array had more tests than the *t*=5-way covering array (3837 vs. 3607) but many fewer than the *t*=6-way covering array (11018 tests). The RBCA *t*=5-way array was much less successful at triggering faults, because of the small number of tests in the test set (1574 tests), which is more comparable to the size of a *t*=4-way covering array (1215 tests).

The initial *t*=4 and *t*=5-way RBCA runs were surprising, catching 23 and 22 bugs, respectively. To investigate why the smaller array was catching more bugs, we changed the order of the equivalence bin values that were presented to the covering array algorithm to generate new arrays. We did this twice for each of the *t*=4 and *t*=5-way arrays, for a total test set of three unique arrays for each *t*-way value. The *t*=4-way arrays triggered 23, 17, and 25 bugs, respectively: an average of 21.7. The *t*=5-way arrays triggered 22, 22, and 25 bugs: an average of 23. At the *t*=6-way interaction size, the RBCA tests were both more effective (catching all bugs across all buggy programs) and efficient (triggering more faults per test input). The RBCA method succeeded by leveraging the strength of covering arrays – i.e., the deliberate use of variable interactions to force the code to traverse statistically unlikely branches – together with the ability of random tests to cover the range of possible values available to the input variables. As is clear from Fig. 1, RBCA catches all of the program bugs with the same number of tests as the *t*=5-way covering array, triggering a fault for the bug that was unreachable by the equivalence bin values used in the simple covering array.

C. *Kolmogorov Complexity-Based Tests*

The set of 10,000 low-complexity inputs generated by the 05AB1E golfing language performed very poorly in triggering the entire range of seeded faults in the 28 TCAS C programs, finding the bug in just one of the 28 programs. However, by measure of efficiency (faults triggered per input) they performed well: 99% of the 10,000 inputs triggered a fault in the same program. The overall efficiency across the 28 programs is 3.5%, which is better than random tests and comparable to covering arrays.

The failure of this test approach to trigger faults in more than one program clearly indicates that it is not useful as a real-world test methodology. Rather than consider complexity as an *absolute* measure of information in the system (the test input), we found it promising to instead investigate complexity as a measure of information distance from a starting input. This approach is consistent with the observation that once a bug is discovered in a program, other bugs are frequently discovered using semantically similar inputs [9].

TABLE 6. RBCA TEST RESULTS

t (strength)	t=2	t=3	t=4	t=5	t=6
Test Size	37	144	476	1334	3837
Bugs Caught	1	13	21.7	23	28
Test Failures	34	154	505	1420	4135
Total Tests	1036	4032	13328	37352	107436
% Efficiency	3.3	3.8	3.8	3.8	3.8

D. *Guided Fuzzing-Based Tests*

Formally, *relative* complexity is the length of the shortest program that generates a desired string given another string as input. Hamming distance is an approximation to this relative complexity, because one such program is the program that modifies the given string by altering a list of specific locations (the length of the program is proportional to the length of this list, i.e., the Hamming distance). There may be shorter programs, but we know this one exists, so it serves as a bound. Thus, the Hamming distance approach can be viewed as a kind of complexity approach, and other extensions are possible. The technique used here to generate new inputs using the Hamming algorithm is intended to be automated and used as a guided-fuzzing tool in future work, which is why we refer to them here as guided fuzzing-based tests.

A form of covering array can also naturally arise from enumerating low-complexity inputs. Per the notation in Kuhn and Okun [21], the system under test has k input variables, each with v possible values. The total number of possible input vectors is v^k, and the number of bits needed to specify an arbitrary input is $k \log v$. Assuming we are interested in t-way interactions with $t << k$, a naïve (non-optimal) t-way covering array can be constructed by taking each configuration of values for each combination of t input variables separately (always assigning a default value to the other $k - t$ variables). The number of t-variable subsets is (k choose t), which for $t << k$ scales with k^t. The number of possible inputs for a given set of t variables is v^t. Thus a covering array can be constructed with size no more than $(kv)^t$. (At the other end, a lower bound on the size is v^t, because this is the minimum number needed to cover *one* set of t variables.)

Likewise, in the complexity approach, there exist short programs that assign values to t variables at a time, starting from a "seed" input string. They do this with some small, fixed algorithm of length C, combined with data indicating which t variables (each of these requires $\log k$ bits) and which values (each of these requires $\log v$ bits). Thus, such a generating program has length (complexity tier) $m = C + t \log(kv)$. Collectively, these programs generate input vectors equivalent to the naïve t-way covering array. And given that C is small and $t << k$, these programs are smaller than the full complexity of arbitrary inputs: $m = C + t \log(kv) << k \log v$. Note that this is a conservative estimate; a more optimized covering array might be present in a complexity tier even lower than m.

TABLE 7. HAMMING TEST RESULTS

	Tier 1	Tier 2	Tier 3
Input Seed Used	**Bugs Caught**	**Bugs Caught**	**Bugs Caught**
UPWARD_RA	13	17	22
UPWARD_RA Tier 1 Output	15	19	22
DOWNWARD_RA	17	22	27
DOWNWARD_RA Tier 1 Output	19	23	27
DOWNWARD_RA Tier 1 Output	17	23	27

Seeding a Hamming algorithm with inputs that traversed low-probability paths through the TCAS code (such as the *DOWNWARD_RA*-output-producing inputs) was highly effective and efficient at triggering the range of seeded bugs in the test programs. This method also suffers from the same drawbacks as the covering array-based approach, in that it uses values from equivalence bins that do not contain the necessary resolution to find small variable value errors in the code. An improved approach would draw random values from within the equivalence bins, similar to RBCA.

The tests generated using the Hamming algorithm were efficient and effective at catching bugs in the faulty C programs, as is apparent from the results in Fig. 1, although they suffered from the same lack of reach as the covering array inputs because they used the equivalence bin values rather than the full range of possible values used in RBCA.

The *DOWNWARD_RA* seeded tests in particular were extraordinarily effective at triggering faults in all of the C programs (except for the bug discussed previously that cannot be reached by the tests because of the choice of equivalence bin values), especially considering the small test size of 2988 inputs. This test size lies in between the t=5 and t=6-way RBCA test set sizes, and is smaller than the t=5-way covering array test set size. Table 7 shows the results of tests using the Hamming-based input test sets. The three DOWNWARD_RA-seeded Tier 3 test sets caught 27/28 bugs, while the Tier 3 test sets that were seeded with UPWARD_RA-generating inputs caught 22/28 bugs.

These results demonstrate a promising approach to guided fuzzing based on relative complexity, which we hope to implement and automate in future research.

ACKNOWLEDGEMENT

The authors would like to thank Richard Kuhn at NIST for providing the TCAS C module and NuSMV model. We thank the Johns Hopkins DEI Collective for providing this research with an outstanding student researcher. Finally, we are grateful to IDA leadership for supporting our work. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

V. REFERENCES

[1] P. Baker, "Software bugs found to be cause of Toyota acceleration death," *San Diego Source,* 2013.

[2] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski and L. K. Rierson, *A practical tutorial on modified condition/decision coverage,* NASA, 2001.

[3] M. Staats, G. Gay, M. Whalen and M. Heimdahl, "On the danger of coverage directed test case generation," in *Fundamental Approaches to Software Engineering*, Tallinn, Estonia, 2012.

[4] RTCA, Incorporated, "DO-178B, Software considerations in airborne systems and equipment certification," 1992.

[5] Wikipedia, "Modified condition/decision coverage," [Online]. Available: https://en.wikipedia.org/wiki/Modified_condition/decision_coverage. [Accessed April 2022].

[6] A. Murugesan, M. P. Whelan, N. Rungta, O. Tkachuk, S. Person, M. P. Heimdahl and D. You, "Are we there yet? Determining the adequacy of formalized requirements and test suites," in *NASA Formal Methods 7th International Symposium Proceedings*, Pasadena, CA, 2015.

[7] M. P. Heimdahl, D. George and R. Weber, "Specification test coverage adequacy criteria = specification test generation inadequacy criteria?," in *IEEE Symposium on High Assurance Systems Engineering*, Tampa, FL, 2004.

[8] A. Murugesan, M. W. Whalen, N. Rungta, O. Tkachuk, S. Person, M. P. Heimdahl and D. You, "Are We There Yet? Determining the Adequacy of Formalized Requirements and Test Suits," in *NASA Formal Methods Symposium*, Pasadena, CA, 2015.

[9] J. R. Mayo and R. C. Armstrong, "Tradeoffs in targeted fuzzing of cyber systems by defenders and attackers," in *Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research*, Oak Ridge, TN, 2011.

[10] RTCA, Incorporated and EUROCAE, "DO-178C, Software considerations in airborne systems and equipment certification," 2012.

[11] K. Fisher, J. Launchbury and R. Richards, "The HACMS program: using formal methods to eliminate exploitable bugs," *Philosophical Transactions of the Royal Society A: Mathematical and Engineering Sciences,* vol. 375, no. 2104, p. 20150401, 2017.

[12] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker and M. Deardeuff, "How Amazon web services uses formal methods," *Communications of the ACM,* vol. 58, no. 4, p. 66, 2015.

[13] P. Ammann and P. E. Black, "Abstracting formal specifications to generate software tests via model checking," in *Gateway to the New Millennium: 18th IEEE Digital Avionics Systems Conference*, St. Louis, MO, 1999.

[14] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *7th European Software Engineering Conference/7th ACM SIGSOFT Symposium*, Amsterdam, 1999.

[15] S. Rayadurgam and M. P. Heimdahl, "Coverage based test case generation using model checkers," in *Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Sytems*, Washington D.C., 2001.

[16] D. C. Montgomery, Design and analysis of experiments, New York: Wiley, 1984.

[17] P. Ammann and J. Offutt, Introduction to Software Testing, Cambridge: Cambridge University Press, 2016.

[18] 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,* vol. 23, no. 7, pp. 437-444, 1997.

[19] M. Zalewski, "American fuzzy lop," Github, [Online]. Available: https://github.com/google/AFL. [Accessed April 2022].

[20] P. Ohlert, "Violating assumptions with fuzzing," *IEEE Security & Privacy,* vol. 3, no. 2, pp. 58-62, 2005.

[21] D. R. Kuhn and V. Okun, "Pseudo-exhaustive testing for software," in *30th Annual IEEE/NASA Software Engineering Workshop*, Columbia, MD, 2006.

[22] M. Staats, "On the Danger of Coverage Directed Test Case Generation".

[23] M. Li and P. M. B. Vitanyi, "Kolmogorov complexity and its applications," in *Handbook of Theoretical Computer Science (Vol. A)*, Cambridge, MA, MIT Press, 1991, pp. 187-254.

[24] Adriandmen, "05AB1E: A Concise Stack-Based Golfing Language," [Online]. Available: https://github.com/Adriandmen/05AB1E.

[25] M. Zalewski, "American Fuzzy Lop," GitHub.

[26] M. Bohme, M. D. Nuguyen, V. T. Pham and A. Roychoudhury, "Directed greybox fuzzing," in *CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas, TX, 2017.