BMC+Fuzz : Efficient and Effective Test Generation

Ravindra Metta
TCS Research
Pune, India
r.metta@tcs.com

Raveendra Kumar Medicherla
TCS Research
Bangalore, India
raveendra.kumar@tcs.com

Samarjit Chakraborty
Department of Computer Science
University of North Carolina at Chapel Hill
samarjit@cs.unc.edu

***Abstract*—Coverage Guided Fuzzing (CGF) is a greybox test generation technique. Bounded Model Checking (BMC) is a whitebox test generation technique. Both these have been highly successful at program coverage as well as error detection. It is well known that CGF fails to cover complex conditionals and deeply nested program points. BMC, on the other hand, fails to scale for programming features such as large loops and arrays.**

To alleviate the above problems, we propose (1) to combine BMC and CGF by using BMC for a short and potentially incomplete unwinding of a given program to generate effective initial test prefixes, which are then extended into complete test inputs for CGF to fuzz, and (2) in case BMC gets stuck even for the short unwinding, we automatically identify the reason, and rerun BMC with a corresponding remedial strategy. We call this approach as *BMCFuzz* and implemented it in the VeriFuzz framework. This implementation was experimentally evaluated by participating in Test-Comp 2021 and the results show that BMCFuzz is both effective and efficient at covering branches as well as exposing errors. In this paper, we present the details of BMCFuzz and our analysis of the experimental results.

I. Introduction

To deal with the rapidly increasing code complexity of software systems like 5G and Machine Learning, embedded software projects are adopting agile processes for large software development, especially DevOps [1]. These processes are assisted by high-degree of automation methodology of Continuous Integration(CI), Continuous Deployment and/or Continuous Delivery (CD). These connected processes, often referred to as "CI/CD pipeline", automate the entire software development life cycle. A key requirement for DevOps is to integrate testing into the software build process itself.

Fuzz testing [2] is an automated technique for detecting defects in functionality as well as security vulnerabilities. It works by fuzzing (mutating) a given corpus of tests to generate more tests to achieve test objectives such as code coverage. This fits in well with testing requirements in the CI/CD pipeline. In particular, Coverage Guided Fuzz (CGF) is an effective testing technique for DevOps.

Another key requirement of DevOps is quick turn around time for testing. Sometimes automated test engines are run once every few minutes. However, CGF performs better with longer fuzzing time (many hours). Also, CGF's performance depends on program structure and initial seed inputs [3]. It is well known that the coverage of the test inputs generated by a CGF is poor if the Program Under Test (PUT) gets "stuck" along those program paths that have *complex* conditions [4]. For example, if a program path has a condition `x == 3` where `x` is a 32-bit integer input, a CGF engine is likely to fail to generate input values that satisfy this condition as the probability of generating such inputs is quite low, 1 out of 2^{32}. Several approaches were proposed to address this shortcoming [4] in restricted settings.

On the other hand, Bounded Model Checking (BMC) [5] is an automated formal verification technique to automatically check if a given system satisfies a given property within executions whose length is bounded by some integer (*bound*). Bounded model checkers use SAT and SMT solvers for solving *complex* conditions to find deeper defects, which CGF struggles to detect. But, BMC suffers from scalability problems in presence of complex features such as loops with large or unknown bounds and large arrays [6]. In the presence of such features, BMC often does not terminate as any sufficiently large bound to cover deeper parts of the program would be too big for BMC. Further, the BMC often gets stuck or fails due to several other reasons (explained in Sec. III-C).

To overcome the above limitations, we attempt to combine the relative strengths of CGF and BMC in a manner that effectively addresses the demands of frequent and fast testing. Given a program P, we propose to unwind P to a *very short* depth d. We guess a d by analyzing the program complexity of P. We then use BMC for generating a testsuite that covers all the branches of P that are reachable within d. Each test in this suite would be incomplete if P has some more inputs beyond d. So, we extend these incomplete tests into complete tests by first computing the ranges (using [7]) of the remaining inputs beyond d and then randomly generating values within these ranges. We feed the testsuite thus obtained to AFL [8].

Further, if BMC gets stuck or fails while generating the tests even for a small d (like 20) due to complex program features as mentioned above, we automatically figure out the reason and rerun BMC with appropriate remedial action such as treating large arrays as uninterpreted functions (see Sec. III-C). Our testsuite generation technique together with the identification of causes and remedial actions for BMC getting stuck, form the main contributions of this paper. We call this approach as *BMCFuzz* and implemented it in the VeriFuzz 1.2 framework [9], by reimplementing the BMC part of the original VeriFuzz framework of [10], and participated in Test-Comp 2021 [11]. The experimental results of Test-Comp 2021 show that BMCFuzz is both effective (leads to better coverage) and efficient (able to generate the testsuites within a short time).

Paper organization: Sec. II briefly introduces CGF and BMC. Sec. III illustrates the issues with CGF and BMC, and details our approach. Sec. IV presents tool architecture and experimen-

978-3-9819263-6-1/DATE22/©2022 EDAA

tal results. Sec. V discusses related work. Sec VI presents our conclusions and future work.

II. Background

A. Coverage Guided Fuzz (CGF) Testing

CGF is a form of *Search Based Software Testing* (SBST) [12]. It uses *evolutionary algorithms* to generate new test inputs and guides the search towards maximizing code coverage [13]. The evolutionary algorithms heuristically *select* the *best-fit* candidates from a *population*. They generate the offsprings by applying *crossover* and *mutating* operations on the selected population. The newer offsprings are checked for their *fitness* against a given objective. The population evolves by adding each fit offspring to the existing population. In a coverage goal driven SBST, a candidate test-input plays the role of an individual in a population. New test-inputs are generated from an existing test-input (called *seed*) by repeatedly applying mutation operations such as flipping a bit at a random position in the seed. The code coverage obtained by the test run on the new inputs serves as fitness metric for the seed.

State-of-the-art grey-box fuzzers such as AFL [8] use *edge coverage* as their fitness metric. AFL is not only simple to use, but also an effective fuzzer that detected vulnerabilities in several well known libraries. We choose AFL as the fuzzer of our choice in this work.

B. Bounded Model Checking

In contrast to CGF, which tries to heuristically explore the input space using search techniques, a Bounded Model Checker (BMC) employs SAT/SMT techniques to check a given system for a given property. In order to scale better, BMC unwinds the given system only up to a given bound. For programs, this bound denotes the length of execution till which the program is to be analyzed. BMC then translates the unwound part of the system, along with the property to be checked, into a Boolean formula. It then checks if this Boolean formula is satisfiable using SAT/SMT solvers.

If the translated Boolean formula is unsatisfiable, then the system satisfies the property up to the given bound. If the system violates the property within the bound, then BMC produces a test case for which the system violates the property. The system could be a high level model such as a UML model or a state machine, or a program or even an executable binary. The property could be any desired property of interest, such as buffer overflow, absence of deadlock, any timing property etc. In this work, we use the C Bounded Model Checker (CBMC) [14] as CBMC is robust, scalable and successfully used in the verification of a variety of real world embedded systems.

III. Problem Illustration and Solution Approach

A. Problem

The code snippet in Fig. 1 illustrates the shortcomings of both CGF and BMC. We adopted this example from the benchmark *loop-industry-pattern/mod3.c* of Test-Comp benchmarks, contributed by industrial researchers as a challenging task for state of the art verification and testing engines.

```
1 int32 x=input(), y=input();
2 if(y == 0x0123ABCD){ // hard for fuzzer
3  while(*){ // unknown #iterations
4   if(x % 3 == 1) ... // easy for both
5   else ...
6  }                    // hard for BMC
7 int32 z=input();
8 if(z % 3 == 2) ...   // easy for both
```

Fig. 1: Motivating Example

```
x=input(); y=input();
if (*){   // unwinding#1 of the loop
   if(x % 3 == 1) ...
   else ...
} else goto loop_end;
if (*){   // unwinding#2 of the loop
   if(x % 3 == 1) ...
   else ...
} else goto loop_end;
...  // #unwinding = ?   (termination)
...  // 2^#unwinding paths (explosion)
loop_end:
if(x % 3 == 2) ...
```

Fig. 2: BMC : unwinding of loop on line#3 of Fig. 1

This code takes three 32-bit integer inputs (lines 1 and 7): x, y, and z. On line-2, it checks if y is equal to the constant 0x0123ABCD. Then, on line-3, there is a loop that iterates an unknown number of times with an if-condition inside (line-4). Lastly, there is an if-condition on line-7 that can be reached only after the loop terminates.

Now, suppose we need to generate a testsuite to cover all the branches in this code snippet. CGF finds it *hard* to generate the input value 0x0123ABCD for the variable y, as the likelihood of generating this particular value is 1 out of 2^{32}, as y is a 32-bit integer. Therefore, CGF is likely to fail to generate input that can cause the test run to cover the true branch of the condition on line 2. However, given a test input for y that exercises the true branch, CGF easily generates test inputs for x and z that exercise both the true and false branches of the if-conditions on lines 4 and 8, as the probability of generating such inputs is 1/3 for the true branches, and 2/3 for the false branches. Lastly, a fuzzer does not have to deal with loop-termination as it just needs to produce random test inputs, unlike BMC that needs to unwind each loop a *sufficient* number of times.

However, in contrast to CGF, BMC can generate test inputs that cover the true branch on line-2 within a fraction of a second, as it uses a SAT/SMT solver as its backend. However, BMC has to completely unwind the loop (line-3), as shown in Fig. 2, as it needs to model the entire program as a SAT/SMT formula. This poses two key challenges: (1) there is no way to know how many times to unwind a loop in any arbitrary program and hence when to terminate the unwinding, and (2) the number of program paths are exponential in the number of unwindings. For example, in Fig. 2 the if-statements on lines 2 and 6 represent unwindings 1 and 2 of the loop. Notice that there are 3^2 paths from line 2 to line 10 (3 paths through the if-statement on line-2, followed by 3-paths through the if-statement on line-6). So, k-unwindings of the loop will lead to k if-else statements and hence to 3^k paths. Such exponential explosion makes the model checking task intractable. Therefore, even though it is easy for BMC to produce test inputs that cover the true and false branches of the if-condition on line-7 of Fig. 1, it will not be able to produce the test-data due to the termination and path-explosion problems.

This shows that there could be parts of a program that are hard for CGF and parts that are hard for BMC, inhibiting both the techniques from producing complete test inputs. Further, DevOps requires the testing to be done in a short time period.

Therefore, our **problem statement** is: *effectively combine the relative strengths of BMC and CGF to generate a testsuite that achieves better coverage in a practically acceptable time*.

B. Solution Step 1: Good Seed Input Generation for CGF Using BMC

As illustrated in Sec. III-A, complex programming features such as big arrays and *complex conditional* statements cause randomly generated inputs to not satisfy the requisite preconditions leading to poor coverage during fuzzing. However, given an the initial corpus of test inputs (*seeds*) that cover the complex conditions, fuzzing such inputs is more *likely* to generate newer test inputs that achieve better coverage [4]. Therefore, we propose to employ BMC in the following way to generate the initial corpus of seed inputs.

While BMC is effective at solving complex constraints to generate test inputs, it suffers from scalability issues as mentioned earlier, especially in the presence of complex loops and arrays [6]. The benchmark programs in [11] are spread across many categories such as Loops, and Combinations of programs of different complex features. These are specifically crafted by researchers from the academia and the industry to be challenging for formal verification engines as well as test generation engines. So, a direct application of BMC for an entire program does not scale well for these benchmarks and a direct application of CGF will lead to poor coverage.

Therefore, for a given program P, we analyze P's program complexity to guess a *short* depth d, depending on the number of loops, their nesting levels and iteration count. Our guessed bound typically ranges from 2 to 10 for complex programs. Then, using CBMC [14], we generate an under-approximate program P_u by unwinding P only till the execution depth d. If this unwinding is incomplete (i.e. P requires more unwinding), then rest of P will be unreachable. This allows BMC to scale much better to this potentially incomplete, but small P_u. We then use CBMC to produce test input sequences that cover the branches of P_u. Such test inputs cater only to P_u, and not to P (a complete unwinding of P may contain more inputs, such as the input z in Fig. 1). Each of these inputs forms a *valid* prefix of a *complete* test input for P. We denote a set of such prefixes with T_p.

Now, each prefix t_p in in T_p needs to be augmented with a suffix t_s to form a complete test input t for P. For this, we first compute the ranges of each of the missing inputs required for t_s using *k*-path interval analysis [7], as this analysis identifies and merges program paths in such a way that the computed intervals (value ranges) help verifying program properties. This analysis conservatively (over approximately) determines the value ranges of inputs that may reach a given program point. We then randomly choose values within these ranges to extend each t_i in T_p into a complete test t for P. These completed tests form the corpus of initial seed tests for CGF, which then fuzzes these to generate more tests for achieving better coverage.

C. Solution Step 2: Remedying a stuck or failed BMC

We observed that sometimes BMC either gets *stuck* (i.e., takes a *lot* of time, like several hours or days) or fails with some error even for short unwindings if the input program contains a *large* number of complex features (many loops, very large arrays, etc.). For identifying the typical problems why BMC gets stuck or fails, we did a comprehensive study of more than 3000 C benchmarks from the Test-Comp repository. These benchmarks consist of a variety of features such as large loops, large arrays, device drivers, heap and bit operations etc., all of which are used in modern embedded software. They were crafted to be challenging for formal verification engines.

CBMC, as well as other BMC engines such as ESBMC [15], first translate a given program and a given property of the program to be verified, into a suitable intermediate representation (IR), such as the goto program of Fig. 2, after unwinding the program to a given or automatically determined depth. The IR is then translated into a SAT formula using appropriate bit-encoding for the data in the programs. Further appropriate constraints are also added to faithfully represent the program semantics, both during the translation as well as during a post-translation processing stage. If BMC does not get stuck or fail during these stages, then the SAT (SMT) formula gets successfully generated. The BMC then calls an off-the-shelf SAT (SMT) solver to solve the formula. Sometimes, even the solver gets stuck while solving a complex formula. If all goes well, the solver successfully verifies the given property. If the property is violated, then the solver lists the corresponding value assignments to the variables in the formula. BMC then maps these values back to the program inputs, generating a trace that shows why the given property is violated.

Suppose we wish to generate a testsuite that covers each control flow branch of a given program P. For this, BMC models the negation of reachability to each branch as a property to be verified and repeatedly calls a SAT solver to check if each such reachability property can be violated. If this property can be violated, it means that the corresponding branch is reachable and the solver generates a set of inputs (as described above) that cause this branch to be reachable, thus forming a testcase for covering this branch. CBMC has an option "–cover branches", with which CBMC automatically instruments all the branch reachability properties to generate a branch coverage testsuite, and repeatedly calls a solver until the reachability of each branch is verified.

During this process, to the best of our knowledge, there have been no known techniques to predict up front if, where and why BMC gets stuck, or whether and when a solver will terminate. So, in order to understand *where* does BMC typically gets stuck or fails, we ran CBMC many times for different time durations, ranging from 1 minute to 2 hours, and identified the typical translation phases and reasons when BMC gets stuck. We have also identified corresponding remedial strategies to rerun BMC to try to overcome them. These problems and corresponding remedial actions are described below.

1) Often times, when there are a large number of nested loops or recursive calls, BMC takes an enormous amount of time (days) to unwind the program even for the short unwinding depth of 10. In such cases, BMC maybe rerun with an even smaller unwind bound. But, during the re-run BMC may again get stuck in unwinding, indicating that the program complexity is simply too high.
   In such cases, one needs to use program abstraction techniques (e.g. [6]) to reduce the complexity of the program, and run BMC on this abstracted program.
2) BMCs use a fixed number of bits to store addressed objects. If the given program has more number of addressed objects than this limit, then BMC throws up an error saying object-bits are insufficient.
   In this case, BMC has to be rerun with a sufficient number of object-bits, which can be automatically determined by counting the number addressed objects in the program.
3) Some programs contain very large arrays (dimension size around a million or even more). As BMCs try flattening such big arrays into individual bits, required for a SAT solver, would lead to too many bits that the SAT solver cannot realistically solve, So, BMC throws an error that the array size is too large.
   In such cases, instead of translating arrays by flattening them into bits, translate arrays as uninterpreted functions for the SAT solver. Alternatively, instead of SAT back-ends, use SMT solvers such as Z3 [16] that support array theories, and hence the flattening is not required.
4) To ensure functionally consistent translation of a program into a SAT or SMT formula, in cases such as when array indices are accessed using non-constants, techniques like Ackermann expansion (refer to [17]) are employed by BMC, which leads to a quadratic number of constraints to be added, irrespective of whether the translation is into a SAT or SMT formula. This means, if an array has a dimension size of 10000, the corresponding Ackermann expansion will have millions of constraints. The BMC takes a lot of time (days) to add such constraints.
   We do not have a remedy for this. In some restricted settings, it is possible to avoid the quadratic constraints. This needs further investigation by experts in BMC.
5) As described earlier, BMC repeatedly calls a SAT solver for each test coverage goal. If a SAT formula for a particular test goal has lots of complex constraints, the solver takes a lot of time to solve the formula (this also depends on the heuristics built into the solver). Further, as BMC times out waiting for the solver, we will lose the test cases already generated for any earlier test goals. In order to capture at least those tests that are already generated, a trap for the timeout signal should be implemented inside BMC. Whenever a timeout is trapped, simply output all the generated tests before exiting BMC. And then re-run BMC with either a different solver or by excluding the test goal for which the solver got stuck, to try to cover the remaining test goals.

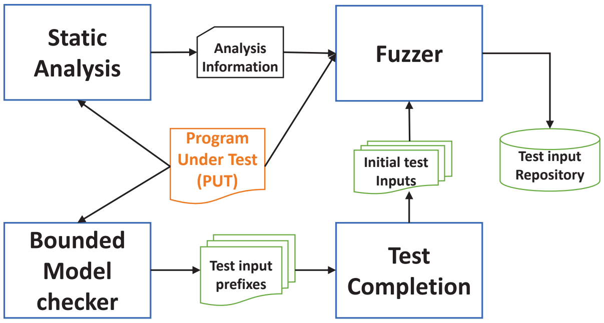

Fig. 3: BMCFuzz Implementation Architecture

Based on our experience in the verification of large embedded systems using BMC, we recommend the above remedies when BMC is employed for the verification of real world embedded systems.

IV. Tool Architecture and Experimental Results

A. Tool Architecture

We have implemented the BMCFuzz approach in VeriFuzz 1.2, available at [9]. Fig. 3 shows the architecture of this implementation.

Given a C language program P ("Program Under Test") as input. P is first fed to a "Bounded Model Checker" (CBMC, in our implementation). CBMC then generates test prefixes as described in III-B. These are then fed to a "Test Completion Engine" (Sec. III-B) that extends the prefixes to form a set of complete test inputs T using the range analysis of [7]. These complete test inputs are then fed to a "Fuzzer". We use an enhanced version of AFL that takes analysis information from a Static Analysis engine, as described in [10]. The Fuzzer mutates each test in T to generate more tests to achieve better coverage.

This implementation currently supports test generation for the two different coverage objectives of Test-Comp 2021: one for covering the branches (*Cover-branches*) and one for covering locations labelled as "ERROR" in the program (*Cover-error*). Lastly, this implementation supports all the remedial strategies except for the integration with program abstraction techniques mentioned in Point 1 of Sec. IV.

B. Experimental Results

In order to comprehensively evaluate the BMCFuzz approach and benchmark it against other state of the art test generation techniques, VeriFuzz 1.2 (i.e. the actual implementation of

BMCFuzz) participated in Test-Comp 2021 [11]. In the rest of this section, "VeriFuzz" refers to "VeriFuzz 1.2".

Test-Comp 2021 consisted of more than 3,000 C programs, each of which consists of a challenging task for verification techniques like BMC as well as test generation techniques like fuzzing. These are classified into several categories as shown in Table I, based on the kind of program features they exercise. Here, the category *ECA* consists of Event-Condition-Action software, *Sequntialized* consists of concurrent programs that have been sequentialized, *xcsp* consists of programs from the XCSP_to_C tool benchmark set, *Combinations* consists of programs combined from other categories, and *Busybox* consists of programs from the Busybox software. The rest of the category names are self explanatory. In Test-Comp 2021, all participating tools are evaluated on these benchmarks with the limits of 15 GB of RAM and 15 minutes of CPU time, on Intel Xeon E3-1230 v5 3.4 GHz CPU, and the results of this evaluation are publicly available [11].

In our experience, except on *pathological programs*, BMC gets stuck only for a couple of reasons, but not all the reasons described in Sec. III-C. Therefore, given the 15 minutes time, in order to quickly detect when BMC gets stuck and rerun with a corresponding remedial strategy, we invoke CBMC for one minute initially and, if CBMC gets stuck, allow maximum two reruns with 1 minute per rerun. After this, we run AFL for the remaining time.

Here, we present our analysis of Test-Comp 2021's tool evaluation results, by comparing VeriFuzz with the other tools that combined fuzzing and symbolic evaluation or BMC: LibKluzzer [18] and Symbiotic [19], two top tools for branch coverage, and (2) with FuSeBMC [20] and LibKluzzer, two top tools in the Cover-Error category.

Table Ia shows the experimental results on 2,565 programs meant for testing branch coverage. In this table, the column #Tasks denotes the number of programs in the corresponding category. For each program P, the evaluation score for a test engine is computed as *the number of branches covered by the test engine / total number of branches in* P. For each tool, Column #s captures the cumulative score obtained by summing up the individual score on all the programs of a category for each test-engine. Column #t denotes the time taken in minutes for producing testsuites for the entire category.

The results show that VeriFuzz achieved better coverage in 9 out of 14 categories when compared to the other tools. In the remaining 5 categories, it is the second best tool with only a narrow difference with the best tool in that category. In particular, in the categories Loops and Combinations, VeriFuzz scored significantly higher than the others. This shows that our combination of BMC and Fuzzing is *effective*, leading to higher coverage. Further, we have also analyzed some of the programs in categories such as xcsp where VeriFuzz did not do that well. Our analysis revealed that, in some cases, CBMC could generate only very few seeds as the time limit of 1 min for rerun was insufficient, VeriFuzz would score more if BMC is run for longer than one minute. In some other cases, the tests produced did not lead to desired coverage due to control dependencies on uninitialized variables, which are treated by CBMC as non-deterministic whereas they lead to undefined execution behaviour in ISO C. We have also observed that in certain programs having floating point inputs, the tests generated by CBMC did not lead to intended coverage due to difference in the interpretation of floating point values between CBMC and AFL. Lastly, VeriFuzz took more time than the others as the fuzzing phase as it keeps on trying to produce test inputs even when a branch is infeasible.

Table Ib shows the experimental results on 594 programs meant for testing error coverage. In each of these programs, only one location is marked as an ERROR location. Each test engine is given a score of 1 if it can produce a test input that causes the program's execution to reach the ERROR location, and 0 otherwise. In Table Ib, the column #s denotes the score of the test engine by summing up its scores for all the individual programs in the corresponding category.

Note that VeriFuzz fared better than the other tools in 2 out 10 categories, and stood second in 7 out of remaining 8 categories. It could locate the error in far lesser time than the other tools. VeriFuzz took 154 sec to find the errors, whereas FuSeBMC consumed 1309 sec and Libkluzzer consumed 5386 sec. In the categories xcsp and recursive, it fared worse. This is for the same reason as in the Cover-branches experiment; these categories just needed CBMC to be run for a bit more time to generate seed inputs.

V. Related work

Combination BMC and Fuzzing has been done earlier. FuSeBMC [20] injects labels, corresponding to desired test coverage goals, to guide BMC and Fuzz engines independently to achieve the desired test objectives. It intelligently manages the execution time of the engines for improving energy consumption. Driller [4] finds security vulnerabilities using a smart combination of fuzzing and a selective concolic execution. Here, when a fuzzer gets ''stuck" in a hard path, symbolic execution techniques have been used to cover the *hard* branch. [21] eliminates such *hard* branches during the fuzzing run to focus only on what could be easily covered.

Another test generation engine that attempted combining BMC and Fuzzing is VeriFuzz 1.0 [10], which employs BMC only to analyze concurrent programs that are sequentialized. Given a sequentialized C program P, it guesses a bound that is sufficiently large to generate (exactly) one complete test input t, such that t executes P till the program-exit point. It then passes t to AFL [8] to generate new test cases. BMC often fails to generate such a t as the guessed bound is often either insufficient or too large for the BMC to scale.

Other tools that use a combination of techniques include LibKluzzer [18], which combines the strengths of coverage-guided fuzzing and whitebox fuzzing, and Symbiotic [19], which integrates light-weight static analyses with program slicing and symbolic execution.

In contrast to all the above, our approach works by splitting a given program into a prefix (a short unwinding of the input program) and a suffix (rest of the program that is not unwound). None of the above split the given program. We first

(a) Cover-Branches Results

Category	#Tasks	VeriFuzz		Libkluzzer		Symbiotic	
		#s	#t	#s	#t	#s	#t
Arrays	400	295	5833	**296**	3000	228	5167
Bitvectors	62	38	933	**38.6**	933	37	600
ControlFlow	67	**18.5**	983	16.1	300	18	63
ECA	29	**11.7**	417	10.1	433	10	367
Floats	226	**98.7**	3333	90.2	3333	50	250
Heap	143	85.7	2167	**89.8**	1833	84	1250
Loops	581	**424**	8667	419	7667	383	4667
Recursive	53	35.1	783	35.9	750	**38**	717
Sequentialized	82	**71.3**	1233	55.1	1217	36	567
xcsp	119	88	1833	80.3	1833	**93**	1833
combinations	210	**180**	3167	139	3167	135	2833
Busybox	72	**8.33**	1083	6.39	1017	7	1000
DeviceDrivers	290	57	4167	**57.8**	3833	44	4000
Termination	231	**204**	3500	199	2000	178	3167
Total	2565	**1615.33**	38099	1533.29	31316	1341	26481

(b) Cover-Error Results

#Tasks	VeriFuzz		FuSeBMC		LibKluzzer	
	#s	#t	#s	#t	#s	#t
100	95	8	93	317	**96**	483
10	9	3	**10**	23	9	135
32	9	6	8	16	**11**	125
18	**16**	27	8	23	11	165
33	30	8	**32**	75	30	450
57	**47**	5	45	38	47	483
158	136	48	131	533	**138**	2000
20	13	5	**19**	15	17	250
107	99	35	**101**	217	83	1250
59	25	9	**53**	52	3	45
NA	NA	NA	NA	NA	NA	NA
NA	NA	NA	NA	NA	NA	NA
NA	NA	NA	NA	NA	NA	NA
NA	NA	NA	NA	NA	NA	NA
594	479	154	**500**	1309	445	5386

TABLE I: Experimental evaluation. For each subcategory, the number in blue colour indicates the highest achieved score.

generate tests for this prefix using BMC, which we extend into tests for the entire program. These tests form a initial corpus of *good* tests for CGF to effectively to produce tests that achieve better coverage. Further, if the BMC gets stuck or fails, our approach automatically identifies the cause and applies appropriate remedial strategies.

VI. Conclusions and Future Work

The experimental results clearly show that the proposed combination of BMC and Fuzz is both effective and efficient.

In future, we plan to integrate our tool with program abstraction techiques. Currently, our approach decomposes a program into a prefix and suffix, and solves them respectively using BMC and Fuzzing. Similar to the compositional BMC of [22], we intend to generalize our approach by decomposing a given program P into n distinct parts (say P_1, P_2, ..., P_n) such that each P_i may be solved with a suitable test generation technique. This will help achieve better scalability and coverage on more complex programs, such as those with several deep loops (e.g., computations on multi-dimensional arrays) interspersed with constraints that are hard for fuzzers to solve.

References

[1] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles, “A survey of devops concepts and challenges,” *ACM Comput. Surv.*, vol. 52, no. 6, Nov. 2019.

[2] V. J. M. Manès, 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*, pp. 1–1, 2019.

[3] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating fuzz testing,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2123–2138.

[4] N. Stephens, J. Grosen *et al.*, “Driller: Augmenting fuzzing through selective symbolic execution,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[5] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Springer, 2018.

[6] S. Kumar, A. Sanyal, R. Venkatesh, and P. Shah, “Property checking array programs using loop shrinking,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Cham: Springer International Publishing, 2018, pp. 213–231.

[7] S. Kumar, B. Chimdyalwar, and U. Shrotri, “Precise range analysis on large industry code,” in *Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013, pp. 675–678.

[8] M. Zalewski, “American fuzzy lop.” [Online]. Available: http://lcamtuf.coredump.cx/afl/

[9] R. Medicherla, “Verifuzz 1.2.0.” [Online]. Available: https://gitlab.com/sosy-lab/test-comp/archives-2021/-/blob/master/2021/verifuzz.zip

[10] A. B. Chowdhury, R. K. Medicherla, and R. Venkatesh, “Verifuzz: Program aware fuzzing - (competition contribution),” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, vol. 11429. Springer, 2019, pp. 244–249.

[11] D. Beyer, “Status report on software testing: Test-comp 2021,” in *Fundamental Approaches to Software Engineering (FASE)*, ser. Lecture Notes in Computer Science, vol. 12649. Springer, 2021, pp. 341–357.

[12] M. Harman, S. A. Mansouri, and Y. Zhang, “Search-based software engineering: Trends, techniques and applications,” *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.

[13] P. McMinn, “Search-based software testing: Past, present and future,” in *International conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2011, pp. 153–163.

[14] D. Kroening and M. Tautschnig, “Cbmc – c bounded model checker,” in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2014, pp. 389–391.

[15] J. Morse, M. Ramalho, L. C. Cordeiro, D. A. Nicole, and B. Fischer, “ESBMC 1.22 - (competition contribution),” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, vol. 8413. Springer, 2014, pp. 405–407.

[16] L. M. de Moura and N. Bjørner, “Z3: an efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008, pp. 337–340.

[17] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, A. Santuari, and R. Sebastiani, “To ackermann-ize or not to ackermann-ize? on efficiently handling uninterpreted function symbols in smt (ut),” in *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Springer-Verlag, 2006, p. 557–571.

[18] H. M. Le, “Llvm-based hybrid fuzzing with libkluzzer (competition contribution),” in *Fundamental Approaches to Software Engineering (FASE)*. Springer International Publishing, 2020, pp. 535–539.

[19] M. Chalupa, J. Novák, and J. Strejček, “Symbiotic 8: Parallel and targeted test generation,” in *Fundamental Approaches to Software Engineering*. Springer International Publishing, 2021, pp. 368–372.

[20] K. M. Alshmrany, M. Aldughaim, A. Bhayat, and L. C. Cordeiro, “Fusebmc: An energy-efficient test generator for finding security vulnerabilities in C programs,” in *Tests and Proofs TAP*, ser. Lecture Notes in Computer Science, F. Loulergue and F. Wotawa, Eds., vol. 12740. Springer, 2021, pp. 85–105.

[21] H. Peng, Y. Shoshitaishvili, and M. Payer, “T-fuzz: Fuzzing by program transformation,” in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 697–710.

[22] C. Y. Cho, V. D’Silva, and D. Song, “Blitz: Compositional bounded model checking for real-world programs,” in *International Conference on Automated Software Engineering ASE*, ser. ASE’13. IEEE Press, 2013, p. 136–146.