


VeriFuzz 1.4: Checking for (Non-)termination (Competition Contribution)

Ravindra Metta, Prasanth Yeduru, Hrishikesh Karmarkar, and Raveendra Kumar Medicherla* (✉)

TCS Research, Tata Consultancy Services, Pune, India
{r.metta,prasanth.yeduru,hrishikesh.karmarkar,raveendra.kumar}@tcs.com

Abstract. In VeriFuzz 1.4, we implemented two new techniques for checking Non-termination and Termination. VeriFuzz 1.4 won the Termination category of SV-COMP 2023.

1 Approach for Non-termination and Termination

VeriFuzz 1.2.0 [4,10,11] is a framework to automatically generate test cases, and lacks the ability to prove properties such as *termination*. Given a program P and *termination* as the property, a tool needs to either provide a witness for Non-termination of P, or give a *true* verdict if P always terminates. Therefore, we developed two techniques: one for *proving* Non-termination and one for checking termination with a *high confidence*, which are described below.

1.1 Technique for Non-termination Checking

For SV-COMP 2023, we implemented a variant of FuzzNT [7], a sound technique for proving Non-termination arising due to infinite loops. FuzzNT takes as input a C program P and a corpus of test inputs T generated using the Coverage Guided Fuzzer of VeriFuzz 1.2. Each test input $t \in T$ is a sequence of values to be supplied to P via *nondet()* calls. We illustrate the key steps of FuzzNT using the program P (Listing 1.1), adopted from the code that caused the SSL non-termination [13]. Note that P terminates on the test input $t = \langle 1 : j = 129, 4 : i == 1, 5 : j = 5, 4 : i == 3 \rangle$. Given such a test input, FuzzNT transforms P into a Path Specific Program (PSP) P' (Listing 1.2), by replacing each nondet() call in P with the corresponding value in the test input, if any, as described in [7]. If multiple values in the test input correspond to a nondet() call in P, FuzzNT picks the first value among them to replace the nondet() call. For example, in t, both $i == 1$ and $i == 3$ correspond to the nondet() call on Line 4 in Listing 1.1. So, as shown on Line 4 of Listing 1.2, this nondet() call is replaced with $i == 1$. Notice that P' has only one feasible execution path, which does not terminate. P' is then supplied to an abstract interpretation based safety checker, which checks if P' does not terminate. If the check succeeds, then P' is non-terminating and

* Jury member

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13994, pp. 594–599, 2023.
https://doi.org/10.1007/978-3-031-30820-8_42

Listing 1.1. Program P

```
1 i=1; j=nondet();
2 while(j!=1) {
3     i=i+1;
4     if (i==nondet()) exit(0);
5     j=nondet(); }
```

Listing 1.2. Program P'

```
1 i=1; j=129;
2 while(j!=1) {
3     i=i+1;
4     if (i==1) exit(0);
5     j=5; }
```

hence P is also non-terminating, and a proof of Non-termination is generated for P in the form of a witness automaton. These steps are repeated until either a non-terminating execution is discovered, or test inputs are exhausted.

1.2 Variant of FuzzNT implemented in VeriFuzz 1.4

The version of FuzzNT in [7] uses Frama-C [14] for the abstract interpretation based Non-termination check. However, we noticed that Frama-C's abstract interpretation does not precisely model termination semantics of standard library functions like *abort()*. This leads to Frama-C incorrectly identifying some terminating programs as non-terminating. Further, we could not bundle Frama-C with FuzzNT due to the installation dependencies and it is unavailable in the Competition Environment of SV-COMP 2023. Therefore, we implemented a variant of FuzzNT using the C Bounded Model Checker [5], as described below.

Given a program P, we begin by checking if P terminates as described in Section 1.3. If this check could not identify the termination of P, then we generate PSPs for P using VeriFuzz 1.2 (described in Section 1.1). Next CBMC is run on each generated PSP, say P', with a small loop unwind bound, say k, and check for CBMC's built-in unwinding assertion, which checks if all loops within P' iterate at most k times. If this check succeeds, then P' is a terminating program. If this check fails, then there exists an input for which some loop in P' iterates more than k times. We then iteratively increase k and repeat the termination check until a large enough k such as $10,000$. In our experiments, we observed that while CBMC does not scale to such a large unwinding of P, it does scale to large unwindings of the PSPs of P, as they admit much fewer behaviours than P. If the check fails even at $10,000$ for P', it is likely to be non-terminating. We then generate a witness automaton for P using P', classifying P as non-terminating.

1.3 Technique for termination

To check if a given a program P terminates on all inputs, we designed an unsound, but high confidence, incremental verification technique based on Bounded Model Checking (CBMC). This technique works in two phases. *Phase-1* is the same as CBMC's own termination check. In this, we begin by unwinding all the loops in P for a small number of iterations, such as 2. Then, using CBMC's built-in loop unwinding assertion check, we verify if all loops terminate within this small unwinding, say k. If this check is successful, then all loops in P terminate within k iterations and hence P itself terminates, and we return *TRUE*

to declare P to be terminating. If the check fails for any loop, then that loop can iterate more than k times. So, we increment k, and repeat the check. This approach suffers from two limitations. (1) As k grows larger, BMC suffers from scalability issues, and (2) if P has a feasible non-terminating path, then the check for a higher k repeats forever. To overcome these limitations, we stop *Phase-1* and return *UNKNOWN* as soon as k reaches a threshold value (pre-configured for SV-COMP 2023). We then proceed to *Phase-2*, described below.

In *Phase-2*, we try to find a small model for the termination property of P, by *guessing* a small range R of the inputs (viz. nondet() calls), such that if P terminates for all inputs in R, then P is *highly likely* to terminate for all its inputs. To guess this R, we learnt a Decision Tree (DT) model on a training data of less than 10% of SV-COMP benchmarks, based on program features and sample execution traces. We are working on formalizing this approach via ranking functions [6].

We then run the incremental verification from *Phase-1*, but by bounding the nondet() values to those in R. This bounding allows CBMC's backend solvers such as Z3 to scale to a larger loop unwind K ($\sim 100,000$ in our experiments). If all loops in P terminate within at most K iterations given the R-bounding, then we assume that P is *highly likely* to terminate on all inputs even without R-bounding. Therefore, if this *bounded value check* concludes that P terminates, then we return *TRUE* to declare P to be terminating, else we return *UNKNOWN* and invoke the non-termination check described in Section 1.2.

2 Software Architecture

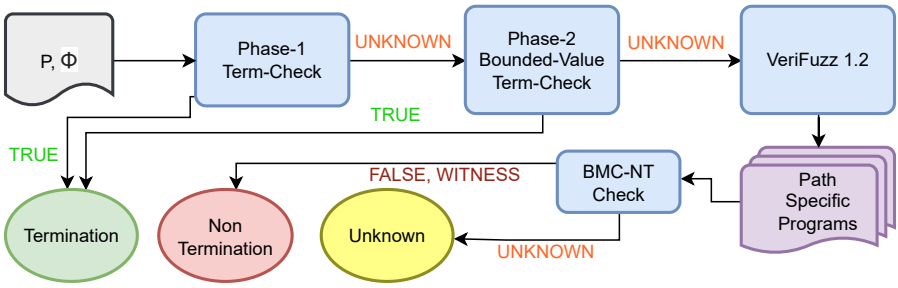

Fig. 1. VeriFuzz 1.4 architecture

Figure 1 shows the architecture of VeriFuzz 1.4. Here P is the input program, and ϕ is the termination property. The process-blocks *Phase-1 Term-Check* and *Phase-2 Bounded-Value Term-Check*, together constitute our two phased termination check described in Section 1.3. If both Phase-1 and Phase-2 return UNKNOWN, we then execute the Non-termination check described in Section 1.2. That is, we first generate PSPs using *VeriFuzz 1.2*, and search for a likely non-terminating PSP, say P'. If we find such a P', we generate a witness automaton and return *FALSE* (to report non-termination). Else, all the above

steps must have returned *UNKNOWN*, and VeriFuzz 1.4 is unable to decide if P is terminating or non-terminating, and hence returns *UNKNOWN*.

In Figure 1, VeriFuzz 1.2 is built using PRISM [8] program analysis framework, AFL [16], and CBMC v5.67.0 [1] with Z3 4.8.15 [12] and Glucose Syrup [2] as the backend SMT and SAT solvers respectively. The DT model used in Phase-2 of the termination check (see Section 1.3) is trained offline using booster trees [3]. The rest of VeriFuzz 1.4 is implemented in C++ and Python.

3 Strengths and Weaknesses

Out of 1043 Termination tasks in SV-COMP 2023, our two phase technique correctly solved 865. Some of these, such as termination-crafted/easy2-2.c and termination-dietlibc/atoi.c, contain loops that iterate arbitrarily large number of times. Hence, while BMC fails to conclude their termination, our approach succeeds as it limits the number of loop iterations by restricting the inputs to a small range. Tasks, such as termination-restricted-15/Sunset.c, terminate within the value ranges guessed during Phase-2, but do not terminate for some inputs that lie outside the ranges. Thus, we wrongly reported them to be terminating.

Out of 766 Non-termination tasks, our Non-termination technique correctly solved 351. Of these, tasks such as systemc/pipeline.cil-1.c , have complex control and data dependencies, which could not be solved by approaches such as those in UAutomizer [9] and Symbiotic [15]. But, the PSPs of these programs, generated by our technique, were much simpler to check for non-termination and hence our technique succeeded on them. However, within the given time limits, if all the PSPs we generated happen to be terminating, then our technique fails to identify the non-termination. Our results on tasks locks/test_locks_14-2.c and termination-restricted-15/Ex02.c demonstrate this behaviour. Another weakness is that our technique currently does not handle programs with recursion.We are currently developing new techniques that address these weaknesses.

4 Tool Configuration and Setup

VeriFuzz 1.4 is available at git@gitlab.com:sosy-lab/sv-comp/archives-2023.git. To install and run the tool, follow the instructions in the `README.txt`. The benchexec tool-info module is `verifuzz.py` and the benchmark definition file is `verifuzz.xml`. A sample run command is as follows: `./scripts/verifuzz.py --propertyFile` *`termination.prp example.c`*. In SV-COMP 2023, VeriFuzz opts to participate in Termination, ReachSafety, and Overflow categories.

5 Software Project and Contributors

VeriFuzz is developed and maintained by the authors at TCS Research. We thank everyone who has contributed to the development of VeriFuzz and the tools AFL, PRISM, CBMC, Glucose Syrup, and Z3. Contact: verifuzz.tool@tcs.com.

6 Data-Availability Statement

VeriFuzz 1.4 is available as part of SV-COMP 2023 verifier repository at https://gitlab.com/sosy-lab/sv-comp/archives-2023/-/blob/main/2023/verifuzz.zip. For any queries, please contact the authors at verifuzz.tool@tcs.com.

References

1. C Bounded Model Checker. https://github.com/diffblue/cbmc
2. Audemard, G., Simon, L.: On the glucose SAT solver. Int. J. Artif. Intell. Tools pp. 1840001:1–1840001:25 (2018). https://doi.org/10.1142/S0218213018400018
3. Chen T., G.C.: Xgboost: A scalable tree boosting system. In: KDD. pp. 785–794 (2016). https://doi.org/10.1145/2939672.2939785
4. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: Verifuzz: Program aware fuzzing - (competition contribution). In: TACAS. pp. 244–249 (2019). https://doi.org/10.1007/978-3-030-17502-3_22
5. Clarke E., Kroening D., L.F.: A tool for checking ansi-c programs. In: TACAS. pp. 168–176 (2004). https://doi.org/10.1007/978-3-540-24730-2_15
6. Giacobbe, M., Kroening, D., Parsert, J.: Neural termination analysis. In: ESEC/FSE. pp. 633–645 (2022). https://doi.org/10.1145/3540250.3549120
7. Karmarkar, H., Medicherla, R., Metta, R., Yeduru, P.: FuzzNT: Checking for program non-termination. In: ICSME. pp. 409–413 (2022). https://doi.org/10.1109/ICSME55016.2022.00049
8. Khare, S., Saraswat, S., Kumar, S.: Static program analysis of large embedded code base: an experience. In: ISEC. pp. 99–102 (2011). https://doi.org/10.1145/1953355.1953368
9. Matthias, H.: Uautomizer (2022), https://gitlab.com/sosy-lab/sv-comp/archives-2022/raw/svcomp22/2022/uautomizer.zip
10. Metta, R., Medicherla, R.K., Chakraborty, S.: BMC+Fuzz: Efficient and Effective Test Generation. In: DATE. pp. 1419–1424 (2022). https://doi.org/10.23919/DATE54114.2022.9774672
11. Metta, R., Medicherla, R.K., Karmarkar, H.: VeriFuzz: Good Seeds for Fuzzing (Competition Contribution). In: FASE. pp. 341–346 (2022). https://doi.org/10.1007/978-3-030-99429-7_20
12. Moura, L.M.d., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
13. OpenSSL: Fix possible infinite loop in BN_mod_sqrt(). https://github.com/openssl/openssl/commit/3118eb64934499d93db3230748a452351d1d9a65 (2022)
14. Patrick, B., et al.: The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. Commun. ACM pp. 56–68 (2021). https://doi.org/10.1145/3470569
15. Viktor, M.: 2LS (2022), https://github.com/diffblue/2ls
16. Zalewski, M.: American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/