Proc. HVC 2017, © Springer

Software Verification: Testing vs. Model Checking
A Comparative Evaluation of the State of the Art

Dirk Beyer and Thomas Lemberger

LMU Munich, Germany

Abstract. In practice, software testing has been the established method for finding bugs in programs for a long time. But in the last 15 years, software model checking has received a lot of attention, and many successful tools for software model checking exist today. We believe it is time for a careful comparative evaluation of automatic software testing against automatic software model checking. We chose six existing tools for automatic test-case generation, namely AFL-fuzz, CPATiger, Crest-ppc, FShell, Klee, and PRtest, and four tools for software model checking, namely Cbmc, CPA-Seq, Esbmc-incr, and Esbmc-kInd, for the task of finding specification violations in a large benchmark suite consisting of 5 693 C programs. In order to perform such an evaluation, we have implemented a framework for test-based falsification (TBF) that executes and validates test cases produced by test-case generation tools in order to find errors in programs. The conclusion of our experiments is that software model checkers can (i) find a substantially larger number of bugs (ii) in less time, and (iii) require less adjustment to the input programs.

1 Introduction

Software testing has been the standard technique for identifying software bugs for decades. The exhaustive and sound alternative, software model checking, is believed to be immature for practice. Some often-named disadvantages are the need for experts in formal verification, extreme resource consumption, and maturity issues when it comes to handling large software systems.

But are these concerns still true today? We claim that the answer is No, and show with experiments on a large benchmark of C programs that software model checkers even find more bugs than testers. We found it is time for a comparative evaluation of testing tools against model-checking tools, motivated by the success of software model checkers as demonstrated in the annual International Competition on Software Verification (SV-COMP) [4], and by the move of development groups of large software systems towards formal verification, such as Facebook [1], Microsoft [2, 44], and Linux [38].

Our contribution is a thorough experimental comparison of software testers against software model checkers. We performed our experimental study on 5 693 programs from a widely-used and state-of-the-art benchmarking set.[2] To represent the state of the art in terms of tools, we use AFL-fuzz, CPATiger,

[1] http://fbinfer.com/ [2] https://github.com/sosy-lab/sv-benchmarks

Crest-ppc, FShell, Klee, and PRTest as software testers, and Cbmc, CPA-Seq, Esbmc-incr, and Esbmc-kInd as software model checkers.[3] The goal in our study is to evaluate the ability to reliably find specification violations in software. While the technique of model checking was originally developed as a proof technique for showing formal correctness, rather than for efficiently finding bugs, this study evaluates all tools exclusively against the goal of finding bugs.

To make the test generators comparable, we developed a unifying framework for test-based falsification (TBF) that interfaces between input programs, test generators, and test cases. For each tester, the infrastructure needs to (a) prepare the input program source code to match the input format that the tester expects and can consume, (b) run the tester to generate test cases, (c) extract test vectors from the tester's proprietary format for the produced test cases, and (d) execute the tests using a test harness to validate whether the generated test cases cover the bug in the program under test (i.e., whether at least one test case exposes the bug). If a bug is found, the framework outputs a witnessing test case in two different, human- and machine-readable formats: (1) a compilable test harness that can be used to directly provoke the bug in the program through execution and (2) a violation witness in a common exchange format for witnesses [7], which can be given to a witness validator to check the specification violation formally or by execution. This allows us to use input programs, produce executable tests, and check program behavior independently from a specific tester's quirks and requirements. We make the following contributions:

- Our framework, TBF, makes AFL-fuzz, CPATiger, Crest-ppc, FShell, Klee, and PRTest applicable to a large benchmark set of C programs, without any manual pre-processing. It is easily possible to integrate new tools. TBF is available online and completely open-source.[4]
- TBF provides two different, human-readable output formats for test cases generated by AFL-fuzz, CPATiger, Crest-ppc, FShell, Klee, and PRTest, and can validate whether a test case describes a specification violation for a program under test. Previously, there was no way to automatically generate test cases with any of the existing tools that are (i) executable and (ii) available in an exchangeable format. This helps in understanding test cases and supports debugging.
- We perform the first comparison regarding bug finding of test-case generation tools and software model checkers at a large scale. The experiments give the interesting insight that software model checkers can identify more program bugs than the existing test-case generators, using less time. All our experimental data and results are available on a supplementary web page.[5]

[3] The choice of using C programs is justified by the fact that C is still the most-used language for safety-critical software. Thus, one can assume that this is reflected in the research community and that the best test-generation and model checking technology is implemented in tools for C. The choice of the particular repository is justified by the fact that this is the largest and most diverse open benchmark suite (cf. SV-COMP [4]).

[4] https://github.com/sosy-lab/tbf [5] https://www.sosy-lab.org/research/test-study/

Related Work. A large-scale comparative evaluation of the bug-finding capabilities of software testers and software model checkers is missing in the literature and this work is a first contribution towards filling this gap. In the area of software model checking, SV-COMP serves as a yearly comparative evaluation of a large set of model checkers for C programs and the competition report provides an overview over tools and techniques [4]. A general survey over techniques for software model checking is available [37]. In the area of software testing, there is work comparing test-case generators [28]. Surveys provide an overview of different test techniques [1] and a detailed web site is available that provides an overview over tools and techniques [6].

2 Background: Technology and Tools

In this paper, we consider only fully automatic techniques for testing and model checking of whole programs. This means that (i) a verification task consists of a program (with function `main` as entry) and a specification (reduced to reachability of function `__VERIFIER_error` by instrumentation), (ii) the comparison excludes all approaches for partial verification, such as unit testing and procedure summarization, and (iii) the comparison excludes all approaches that require interaction as often needed for deductive verification.

2.1 Software Testing

Given a software system and a specification of that system, testing executes the system with different input values and observes whether the intended behavior is exhibited (i.e., the specification holds). A *test vector* $\langle \eta_1, \cdots, \eta_n \rangle$ is a sequence of n input values η_1 to η_n. A *test case* is described by a test vector, where the i-th input of the test case is given by the i-th value η_i of the test vector. A *test suite* is a set of test cases. A *test harness* is a software that supports the automatic execution of a test case for the program under test, i.e., it feeds the values from the test vector one by one as input to the program. *Test-case generation* produces a set of test vectors that fulfills a specific coverage criterion. Program-branch coverage is an example of a well-established coverage criterion.

There are three major approaches to software test-case generation: symbolic or concolic execution [18, 19, 29, 39, 45, 46], random fuzz testing [30, 36], and model checking [5, 10, 35]. In this work, we use one tester based on symbolic execution (Klee), one based on concolic execution (Crest-ppc), one based on random generation (PRtest), one based on random fuzzing (AFL-fuzz), and two based on model-checking (CPATiger and FShell), which we describe in the following in alphabetic order. Table 1 gives an overview over testers and model checkers.
AFL-fuzz [17] is a coverage-based greybox fuzzer. Given a set of start inputs, it performs different mutations (e.g., bit flips, simple arithmetics) on the existing inputs, executes these newly created inputs, and checks which parts of the program are explored. Depending on these, it decides which inputs to keep, and which to use for further mutations. *Output:* AFL-fuzz outputs each generated

[6] Provided by Z. Micskei: http://mit.bme.hu/~micskeiz/pages/code_based_test_generation.html

Table 1: Overview of test generators and model checkers used in the comparison

Tool	Ref.	Version	Technique
AFL-fuzz	[17]	`2.46b`	Greybox fuzzing
Crest-ppc	[39]	`f542298d`	Concolic execution, search-based
CPATiger	[10]	`r24658`	Model checking-based testing, based on CPAchecker
FShell	[35]	`1.7`	Model checking-based testing, based on Cbmc
Klee	[19]	`c08cb14c`	Symbolic execution, search-based
PRtest		`0.1`	Random testing
Cbmc	[40]	`sv-comp17`	Bounded model checking
CPA-Seq	[25]	`sv-comp17`	Explicit-state, predicate abstraction, k-Induction
Esbmc-incr	[43]	`sv-comp17`	Bounded model checking, incremental loop bound
Esbmc-kInd	[27]	`sv-comp17`	Bounded model checking, k-Induction

test case in its own file. The file's binary representation is read 'as is' as input, so generated test cases do not have a specific format.

CPATiger [10] uses model checking, more specifically, predicate abstraction [12], for test case generation. Is is based on the software-verification tool CPAchecker [11] and uses the FShell query language (FQL) [35] for specification of coverage criteria. If CPATiger finds a feasible program path to a coverage criterion with predicate abstraction, it computes test inputs from the corresponding predicates used along that path. It is designed to create test vectors for complicated coverage criteria. *Output:* CPATiger outputs generated test cases in a single text file, providing the test input as test vectors in decimal notation together with additional information.

Crest [18] uses concolic execution for test-case generation. It is search-based, i.e., it chooses test inputs that reach yet uncovered parts of the program furthest from the already explored paths. Crest-ppc [39] improves on the concolic execution used in Crest by modifying the input generation method to query the constraint solver more often, but using only a small set of constraints for each query. We performed experiments to ensure that Crest-ppc outperforms Crest. The results are available on our supplementary web page. *Output:* Crest-ppc outputs each generated test case in a text file, listing the sequence of used input values in decimal notation.

FShell [35] is another model-checking-based test-case generator. It uses CBMC (described in Sect. 2.2) for state-space exploration and also uses FQL for specification of coverage criteria. *Output:* FShell outputs generated test cases in a single text file, listing input values of tests together with additional information. Input values of tests are represented in decimal notation.

Klee [19] uses symbolic execution for test-case generation. After each step in a program, Klee chooses which of the existing program paths to continue on next, based on different heuristics, including a search-based one and one preventing inefficient unrolling of loops. Since Klee uses symbolic execution, it can explore the full state space of a program and can be used for software verification, not just test-case generation. As we are interested in exploring the capabilities of

testing, we only consider the test cases produced by Klee. *Output:* Klee outputs each generated test case in a binary format that can be read with Klee. The input values of tests are represented by their bit width and bit representation. **PRtest** is a simple tool for plain random testing. The tool is delivered together with TBF and serves as base line in our experiments. *Output:* PRtest outputs each generated test case in a text file, listing the sequence of used input values in hexadecimal notation.

2.2 Software Model Checking

Software model checking tries to prove a program correct or find a property violation in a program, by exploring the full state space and checking whether any of the feasible program states violate the specification. A lot of different techniques exist to do this. Since the number of concrete states of a program can be, in general, infinite, a common principle is *abstraction*. A good abstraction is, on the one hand, as coarse as possible —to keep the state space that must be explored small— and, on the other hand, precise enough to eliminate false alarms.

Tools for software model checking combine many different techniques, for example, counterexample-guided abstraction refinement (CEGAR) [21], predicate abstraction [31], bounded model checking (BMC) [16, 22], lazy abstraction [9, 34], k-induction [8, 27], and interpolation [23, 42]. A listing of the widely-used techniques, and which tools implement which technique, is given in the SV-COMP'17 report [4] in Table 4. In this work, we use a general-purpose bounded model checker (Cbmc), a sequential combination of approaches (CPA-Seq), a bounded model checker with incrementally increasing bounds (Esbmc-incr), and a k-induction based model checker (Esbmc-kInd).

Cbmc [22, 40] uses bit-precise BMC with MiniSat [26] as SAT-solver backend. BMC performs model checking with limited loop unrolling, i.e., loops are only unrolled up to a given bound. If no property violation can be found in the explored state space under this restriction, the program is assumed to be safe in general.

CPA-Seq [25] is based on CPAchecker that combines explicit-state model checking [13], k-induction [8], and predicate analysis with adjustable-block abstraction [12] sequentially. CPA-Seq uses the bit-precise SMT solver MathSAT5 [20].

Esbmc-incr [43] is a fork of Cbmc with an improved memory model. It uses an iterative scheme to increase its loop bounds, i.e., if no error is found in a program analysis using a certain loop bound, then the bound is increased. If no error is found after a set number of iterations, the program is assumed to be safe.

Esbmc-kInd [27] uses automatic k-induction to compute loop invariants in the context-bounded model checking of Esbmc. It performs the three phases of k-induction in parallel, which often yields a performance advantage.

2.3 Validation of Results

It is well-understood that when testers and model checkers produce test cases and error paths, respectively, sometimes the results contain false alarms. In order to avoid investing time on false results, test cases can be validated by reproducing a real crash [24, 41] and error paths can be evaluated by witness validation [7, 15]. A *violation witness* is an automaton that describes a set of paths through the

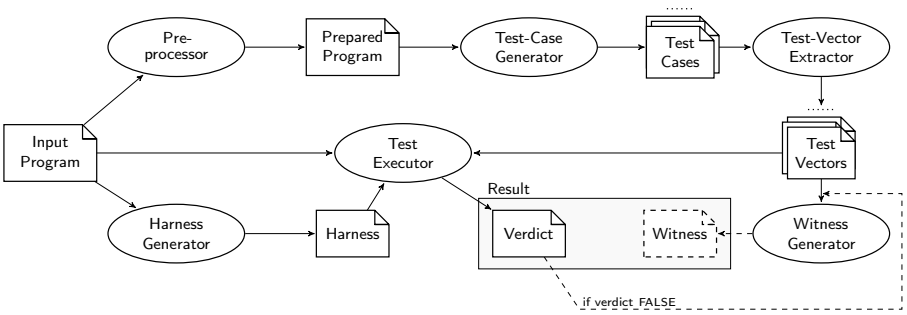

Fig. 1: Workflow of TBF

program that contain a specification violation. Each state transition contains a *source-code guard* that specifies the program-code locations at which the transition is allowed, and a *state-space guard* that constrains the set of possible program states after the transition. We considered four existing witness validators.
CPAchecker [7] uses predicate analysis with adjustable-block abstraction combined with explicit-state model checking for witness validation.
CPA-witness2test[7] creates a compilable test harness from a violation witness and checks whether the specification violation is reached through execution.
FShell-witness2test[8] also performs execution-based witness validation, but does not rely on any verification tool.
Ultimate Automizer [32] uses an automata-centric approach [33] to model checking for witness validation.

In this work, we evaluate the results from testers with TBF by considering for each test case, one by one, whether compiled with a test harness and the program, the execution violates the specification, and we evaluate the results of model checkers by validating the violation witness using four different witness validators. This way, we count bug reports only if they can be reproduced.

3 Framework for Test-Based Falsification

We designed a framework for test-based falsification (TBF) that makes it possible to uniformly use test-case generation tools. Figure 1 shows the architecture of this approach. Given an input program, TBF first pre-processes the program into the format that the test-case generator requires ('prepared program'). This includes, e.g., adding function definitions for assigning new symbolic values and compiling the program in a certain way expected by the generator. The prepared program is then given to the test-case generator, which stores its output in its own, proprietary format ('test cases'). These test cases are given to a test-vector extractor to extract the test vectors and store them in an exchangeable, uniform format ('test vectors'). The harness generator produces a test harness for the input program, which is compiled and linked together with the input program and executed by the test executor. If the execution reports a specification violation, the verdict is FALSE. In all other cases, the verdict

[7] https://github.com/sosy-lab/cpachecker
[8] https://github.com/tautschnig/cprover-sv-comp/tree/test-gen/witness2test

```c
int nondet_int();
short nondet_short();
void __VERIFIER_error();

int main() {
  int x = nondet_int();
  int y = x;

  if (nondet_short()) {
    x++;
  } else {
    y++;
  }

  if (x > y) {
    __VERIFIER_error();
  }
}
```

Fig. 2: An example C program

```c
int nondet_int(){
    int __sym;
    CREST_int(__sym);
    return __sym;
}
```

Fig. 3: A function definition prepared for CREST-PPC

```c
void __VERIFIER_error() {
  fprintf(stderr, "__TBF_error_found.\n");
  exit(1);
}

int nondet_int() {
  unsigned int inp_size = 3000;
  char * inp_var = malloc(inp_size);
  fgets(inp_var, inp_size, stdin);
  return *((int *) parse_inp(inp_var));
}

short nondet_short() {
  unsigned int inp_size = 3000;
  char * inp_var = malloc(inp_size);
  fgets(inp_var, inp_size, stdin);
  return *((short *) parse_inp(inp_var));
}
```

Fig. 4: Excerpt of a test harness; test vectors are passed by standard input (`fgets`, `parse_inp`)

is UNKNOWN. If the verdict of a program is FALSE, TBF produces a self-contained, compilable test harness and a violation witness for the user.

Input Program. TBF is designed to evaluate test-case generation tools and supports the specification encoding that is used by SV-COMP. In this work, all programs are C programs and have the same specification: "Function `__VERIFIER_error` is never called."

Pre-processor. TBF has to adjust the input programs for the respective test-case generator that is used. Each test-case generator uses certain techniques to mark input values. We assume that, except for special functions that are defined by the rules for the repository[9], all undefined functions in the program are free of side effects and return non-deterministic values of their corresponding return type. For each undefined function, we append a definition to the program under test to inject a new input value whenever the specific function is called. The meaning of the special functions defined by the repository rules are also represented in the code. Figure 2 shows a program with undefined functions `nondet_int` and `nondet_short`. As an example, Fig. 3 shows the definition of `nondet_int` that tells CREST-PPC to use a new (symbolic) input value. We display the full code of pre-processed example programs for all considered tools on our supplementary web page. After pre-processing, we compile the program as expected by the test-case generator, if necessary.

Test-Vector Extractor. Each tool produces test cases as output as described in Sect. 2.1. For normalization, TBF extracts test vectors from the generated test cases in an exchangeable format. We do not wait until the test generator is finished, but extract a test vector whenever a new test case is written, in parallel.

[9] https://sv-comp.sosy-lab.org/2017/rules.php

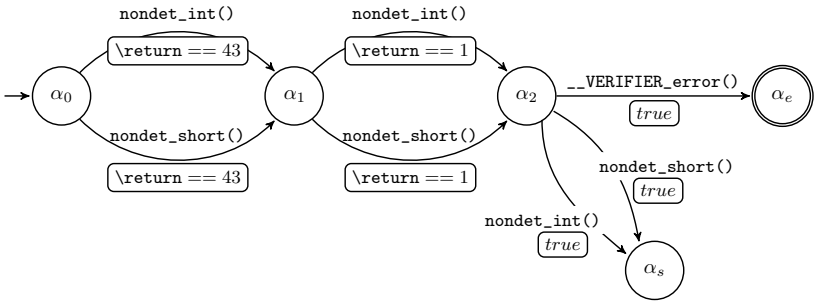

Fig. 5: Violation witness for test vector $\langle 43, 1 \rangle$ and two non-deterministic methods

Harness Generator and Test Executor. We provide an effective and efficient way of checking whether a generated test vector represents a property violation: We create a test harness for the program under test that can feed an input value into the program for each call to a non-deterministic function. For performance reasons, it gets these input values from standard input. For each test vector extracted from the produced test cases, we execute the pre-compiled test harness with the vector as input and check whether a property violation occurs during execution. An example harness is shown in Fig. 4.

Witness Generation. A test vector $\langle \eta_1, \cdots, \eta_n \rangle$ can be represented by a violation witness that contains one initial state α_0, one accepting state α_e, one sink sate α_s, and, for each value η_i of the test vector, a state α_i with, for each non-deterministic function occurring in the program, a transition from α_{i-1} to α_i with the call to the corresponding function as source-code guard and η_i as return value for the corresponding function as state-space guard, i.e.: the transition can only be taken if the corresponding function is called, and, if the transition is taken, it is assumed that the return value of the corresponding function is η_i. From α_n, there is one transition to α_e for each occurring call to `__VERIFIER_error`, and one transition to α_s for each non-deterministic function in the program. Each such transition has the corresponding function call as source-code guard and no state-space guard. The transitions to sink state α_s make sure that no path is considered that may need an additional input value. While such a path may exist in the program, it can not be the path described by the test vector. Fig. 5 shows an example of such a witness. Each transition between states is labeled with the source-code guard (no box) and the state-space guard (boxed). The value '*true*' means that no state-space guard exists for that transition.

When validating the displayed violation witness, a validator explores the state-space until it encounters a call to `nondet_int` or `nondet_short`. Then, it is told to assume that the encountered function returns the concrete value 43, described by the special identifier `\return`. When it encounters one of the two functions for the second time, it is told to assume that the corresponding function returns the concrete value 1. After this, if it encounters a call to `__VERIFIER_error`, it confirms the violation witness. If it encounters a call to one of the two non-deterministic functions for the third time, it enters the sink state α_s, since our witnessed counterexample only contains two calls to non-deterministic functions.

4 Experimental Evaluation

We compare automatic test generators against automatic software model checkers regarding bug finding abilities in a large-scale experimental evaluation.

4.1 Experiment Setup

Programs under Test. To get a representative set of programs under test, we used all 5 693 verification tasks of the SV-BENCHMARKS set[10] in revision `879e141f`[11] whose specification is that function `__VERIFIER_error` is not called. Of the 5 693 programs, 1 490 programs contain a known bug (at most one bug per program), i.e., there is a path through the program that ends in a call to `__VERIFIER_error`, and 4 203 programs are correct. The benchmark set is partitioned into categories. A description of the kinds of programs in the categories of an earlier version of the repository can be found in the literature (cf. [3], Sect. 4). For each category (e.g., 'Arrays'), the defining set of contained programs (`.set` file[12]), and a short characterization and the bit architecture of the contained programs (`.cfg` file[13]) can be found in the repository itself.

Availability. More details about the programs under test, generated test cases, generated witnesses, and other experimental data are available on the supplementary web page.[14]

Tools. We used the test generators and model checkers in the versions specified in Table 1. TBF[15] is implemented in Python `3.5.2` and available as open-source; we use TBF in version `0.1`. For CREST-PPC, we use a modified revision that supports long data types. For readability, we add superscripts T and M to the tool names for better visual identification of the testers and model checkers, respectively. We selected six testing tools that (i) support the language C, (ii) are freely available, (iii) cover a spectrum of different technologies, (iv) are available for 64-bit GNU/Linux, and (v) generate test cases for branch coverage or similar: AFL-FUZZ, CPATIGER, CREST-PPC, FSHELL, KLEE, and PRTEST. For the model checkers, we use the four most successful model checkers in category 'Falsification' of SV-COMP'17[16], i.e., CBMC, CPA-SEQ, ESBMC-INCR, and ESBMC-KIND. To validate the results of violation witnesses, we use CPACHECKER and ULTIMATE AUTOMIZER in the revision from SV-COMP'17, CPA-WITNESS2TEST in revision `r24473` of the CPACHECKER repository, and FSHELL-WITNESS2TEST in revision `2a76669f` from branch `test-gen` in the CPROVER repository[17].

Computing Resources. We performed all experiments on machines with an Intel Xeon E3-1230 v5 CPU, with 8 processing units each, a frequency of 3.4 GHz, 33 GB of memory, and a Ubuntu 16.04 operating system with kernel Linux 4.4.

[10] https://sv-comp.sosy-lab.org/2017/benchmarks.php
[11] https://github.com/sosy-lab/sv-benchmarks/tree/879e141f
[12] https://github.com/sosy-lab/sv-benchmarks/blob/879e141f/c/ReachSafety-Arrays.set
[13] https://github.com/sosy-lab/sv-benchmarks/blob/879e141f/c/ReachSafety-Arrays.cfg
[14] https://www.sosy-lab.org/research/test-study/
[15] https://github.com/sosy-lab/tbf
[16] https://sv-comp.sosy-lab.org/2017/results/
[17] https://github.com/tautschnig/cprover-sv-comp

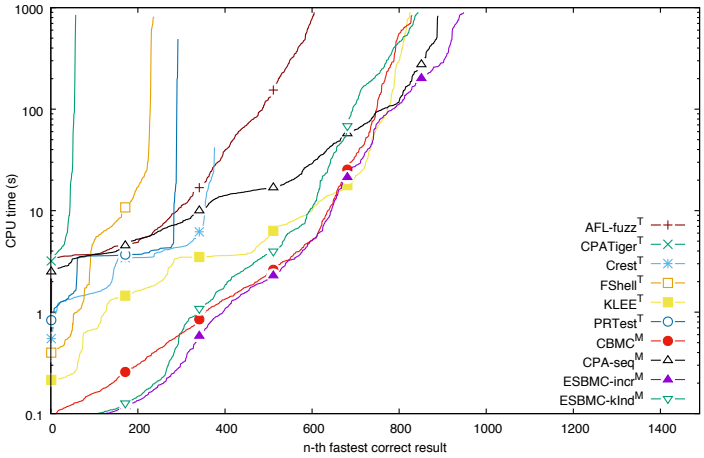

Fig. 6: Quantile plots for the different tools for finding bugs in programs

We limited each benchmark run to 2 processing units, 15 GB of memory, and 15 min of CPU time. All CPU times are reported with two significant digits.

4.2 Experimental Results

Now we report the results of our experimental study. For each of the 1 490 programs that contain a known bug, we applied all testers and model checkers in order to find the bug. For the testers, a bug is found if one of the generated test cases executes the undesired function call. For the model checkers, a bug is found if the tools return answer FALSE together with a violation witness.

Qualitative Overview. We illustrate the overall picture using the quantile plot in Fig. 6. For each data point (x, y) on a graph, the quantile plot shows that x bugs can be correctly identified using at most y seconds of CPU time. The x-position of the right-most data point for a tool indicates the total number of bugs the tool was able to identify. In summary, each model checker finds more bugs than the best tester, while the best tester (KLEE$^{\text{T}}$) closely follows the weakest model checker (CBMC$^{\text{M}}$).

The area below the graph is proportional to the overall consumed CPU time for successfully solved problems. The visualization makes it easy to see, e.g., by looking at the 400 fastest solved problems, that most testers time out while most model checkers use only a fraction of their available CPU time. In summary, the ratio of returned results by invested resources is much better for the model checkers.

Quantitative Overview. Next, we look at the numerical details as shown in Table 2. The columns are partitioned into four parts: the table lists (i) the category/row label together with the number of programs (maximal number of found bugs), (ii) the number of found bugs for the six testers, (iii) the number of found bugs for the four model checkers, and (iv) the union of the results for testers, model checkers, and overall. In the two parts for the testers and model checkers, we highlight the best result in bold (if equal, the fastest result is highlighted). The rows are partitioned into three parts: the table shows first

Table 2: Results for testers and model checkers on programs with a bug

	No. Programs	AFL-FUZZ^{T}	CPATIGER^{T}	CREST-PPC^{T}	FSHELL^{T}	KLEE^{T}	PRTEST^{T}	CBMC^{M}	CPA-SEQ^{M}	ESBMC-INCR^{M}	ESBMC-KIND^{M}	Union Testers	Union MC	Union All
Arrays	81	**26**	0	20	4	22	25	**6**	3	6	4	31	13	33
BitVectors	24	**11**	5	7	5	11	10	12	12	12	**12**	14	17	19
ControlFlow	42	15	0	11	3	**20**	3	**41**	23	36	35	21	42	42
ECA	413	234	0	51	0	**260**	0	143	**257**	221	169	286	42	338
Floats	31	**11**	2	2	4	2	11	**31**	29	17	13	13	31	31
Heap	66	46	22	16	13	**48**	32	**64**	31	62	58	48	66	66
Loops	46	**45**	27	29	5	40	33	**42**	36	42	38	41	38	43
ProductLines	265	169	1	204	156	**255**	144	263	**265**	265	263	265	265	265
Recursive	45	44	0	35	22	**45**	31	**42**	41	40	40	45	43	45
Sequentialized	170	4	0	1	24	**123**	3	**135**	122	135	134	123	141	147
LDV	307	0	0	0	0	0	0	51	70	**113**	78	0	147	147
Total Found	1 490	605	57	376	236	**826**	292	830	889	**949**	844	887	1 092	1 176
Compilable	1 115	605	57	376	236	**826**	292	779	819	**830**	761	887	930	1 014
Wit. Confirmed	1 490							761	**857**	705	634	887	979	1 068
Median CPU Time (s)		11	4.5	**3.4**	6.2	3.6	3.6	**1.4**	15	1.9	2.3			
Average CPU Time (s)		82	38	**4.1**	27	33	6.7	**46**	51	61	69			

the results for each of the 11 categories of the programs under test, second the results for all categories together, and third the CPU times required.

The row 'Total Found' shows that the best tester (KLEE^{T}) is able to find 826 bugs, while all model checkers find more, with the best model checker (ESBMC-INCR^{M}) finding 15 % more bugs (949) than the best tester. An interesting observation is that the different tools have different strengths and weaknesses: column 'Union Testers' shows that applying all testers together increases the amount of solved tasks considerably. This is made possible using our unifying framework TBF, which abstracts from the differences in input and output of the various tools and lets us use all testers in a common work flow. The same holds for the model checkers: the combination of all approaches significantly increases the number of solved problems (column 'Union MC'). The combination of testers and model checkers (column 'Union All') in a bug-finding workflow can further improve the results significantly, i.e., there are program bugs that one technique can find but not the other, and vice versa.

While it is usually considered an advantage that model checkers can be applied to incomplete programs that are not yet fully defined (as expected by static-analysis tools), testers obviously cannot be applied to such programs (as they are dynamic-analysis tools). This issue applies in particular to the category 'LDV' of device drivers, which contain non-deterministic models of the operating-system environment. This kind of programs is important because it is successfully used to find bugs in systems code [18] [47], but in order to provide a comparison without the influence of this issue, we also report the results restricted to those programs that are compilable (row 'Compilable').

[18] http://linuxtesting.org/results/ldv

For the testers, TBF validates whether a test case is generated that identifies the bug as found. This test case can later be used to reproduce the error path using execution, and a debugger helps to comprehend the bug. For the model checkers, the reported violation witness identifies the bug as found. This witness can later be used to reproduce the error path using witness validation, and an error-path visualizer helps to comprehend the bug. Since the model checkers usually do not generate a test case, we cannot perform the same validation as for the testers, i.e., execute the program with the test case and check if it crashes. However, all four model checkers that we use support exchangeable violation witnesses [7], and we can use existing witness validators to confirm the witnesses. We report the results in row 'Wit. Confirmed', which counts only those error reports that were confirmed by at least one witness validator. While this technique is not always able to confirm correct witnesses (cf. [4], Table 8), the big picture does not change. The test generators do not need this additional confirmation step, because TBF takes care of this already. There are two interesting insights: (1) Software model checkers should in addition produce test data, either contained in the violation witness or as separate test vector. This makes it easier to reproduce a found bug using program execution and explore the bug with debugging. (2) Test generators should in addition produce a violation witness. This makes it easier to reproduce a found bug using witness validation and explore the bug with error-path visualization [6].

Consideration of False Alarms. So far we have discussed only the programs that contain bugs. In order to evaluate how many false alarms the tools produce, we have also considered the 4 203 programs without known bug. All testers report only 3 bugs on those programs. We manually investigated the cause and found out that we have to blame the benchmark set for these, not the testers.[19] Each of the four model checkers solves at least one of these three tasks with verdict TRUE, implying an imprecise handling of floating-point arithmetics. The model checkers also produce a very low number of false alarms, the largest number being 6 false alarms reported by ESBMC-INCR^{M}.

4.3 Validity

Validity of Framework for Test-Based Falsification. The results of the testers depend on a correctly working test-execution framework. In order to increase the confidence in our own framework TBF, we compare the results obtained with TBF against the results obtained with a proprietary test-execution mechanism that KLEE provides: KLEE-REPLAY[20]. Figure 7 shows the CPU time in seconds required by KLEE^{T} using TBF (x-axis) and KLEE-REPLAY (y-axis) for each verification task that could be solved by either one of them. It shows that KLEE^{T} (and thus, TBF) is very similar to KLEE's native solution. Over all verification

[19] There are three specific programs in the *ReachSafety-Floats* category of SV-COMP that are only safe if compiled with 64-bit rounding mode for floats or for a 64-bit machine model. The category states the programs should be executed in a 32-bit machine model, which seems incorrect.

[20] http://klee.github.io/tutorials/testing-function/#replaying-a-test-case

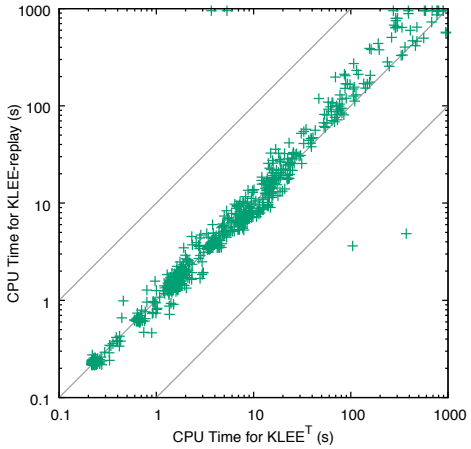

Fig. 7: CPU time required by KLEE^{T} and KLEE-REPLAY to solve tasks

tasks, KLEE^{T} is able to find bugs in 826 tasks, while KLEE-REPLAY is able to find bugs in 821 tasks. There are 15 tasks that KLEE-REPLAY can not solve, while KLEE^{T} can, and 10 tasks that KLEE-REPLAY can solve, while KLEE^{T} can not.

For KLEE^{T}, one unsolved task is due to missing support of a corner case for the conversion of KLEE's internal representation of numbers to a test vector. The remaining difference is due to an improper machine model: for KLEE-REPLAY, we only had 64-bit libraries available, while most tasks of SV-COMP are intended to be analyzed using a 32-bit architecture. This only results in a single false result, but interprets some of the inputs generated for 32-bit programs differently, thus reaching different parts of the program in a few cases. This also explains the few outliers in Fig. 7. The two implementations both need a median of 0.43 s of CPU time to find a bug in a task. This shows that our implementation is similarly effective and efficient to KLEE's own, tailored test-execution mechanism.

Other Threats to Internal Validity. We used the state-of-the-art benchmarking tool BENCHEXEC [14] to run every execution in an isolated container with dedicated resources, making our results as reliable as possible. Our experimental results for the considered model checkers are very close to the results of SV-COMP'17[21], indicating their accuracy. Our framework TBF is a prototype and may contain bugs that degrade the real performance of test-based falsification. Probably more tasks could be solved if more time was invested in improving this approach, but we tried to keep our approach as simple as possible to influence the results as less as possible.

Threats to External Validity. There are several threats to external validity. All tools that we evaluated are aimed at analyzing C programs. It might be the case that testing research is focused on other languages, such as C++ or Java. Other languages may contain other quirks than C that make certain approaches to test-case generation and model checking more or less successful. In addition,

[21] https://sv-comp.sosy-lab.org/2017/results/

there may be tools using more effective testing or model-checking techniques that were developed for other languages and thus are not included here.

The selection of testers could be biased by the authors' background, but we reflected the state-of-the-art (see discussion of selection) and related work in our choice. While we tried to represent the current landscape of test-case generators by using tools that use fundamentally different approaches, there might be other approaches that may perform better or that may be able to solve different tasks. We used most of the recent, publicly available test-case generators aimed at sequential C programs. We did not include model-based or combinatorial test-case generators in our evaluation.

For representing the current state-of-the-art in model checking, we only used four tools to limit the scope of this work. The selection of model checkers is based on competition results: we simply used the four best tools in SV-COMP'17. There are many other model-checking tools available. Since we performed our experiments on a subset of the SV-COMP benchmark set and used a similar execution environment, our results can be compared online with all verifiers that participated in the competition. The software model checkers might be tuned towards the benchmark set, because all of the software model checkers participated in SV-COMP, while of the testers, only FShell participated in SV-COMP before.

While we tried to achieve high external validity by using the largest and most diverse open benchmark set, there is a high chance that the benchmark set does not appropriately represent the real landscape of existing programs with and without bugs. Since the benchmark set is used by the SV-COMP community, it might be biased towards software model checkers, and thus, must stay a mere approximation.

5 Conclusion

Our comparison of software testers with software model checkers has shown that the considered model checkers are competitive for finding bugs on the used benchmark set. We developed a testing framework that supports the easy comparison of different test-case generators with each other, and with model checkers. Through this, we were able to perform experiments that clearly showed that model checking is mature enough to be used in practice, and even outperforms the bug-finding capabilities of state-of-the-art testing tools. It is able to cover more bugs in programs than testers and also finds those bugs faster. With this study, we do not pledge to eradicate testing, whose importance and usability can not be stressed enough. But we laid ground to show that model checking should be considered for practical applications. Perhaps the most important insight of our evaluation is that is does not make much sense to distinguish between testing and model checking if the purpose is finding bugs, but to leverage the strengths of different techniques to construct even better tools by combination.

References

1. S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test-case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

2. T. Ball and S. K. Rajamani. The Slam project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.
3. D. Beyer. Competition on software verification (SV-COMP). In *Proc. TACAS*, LNCS 7214, pages 504–524. Springer, 2012.
4. D. Beyer. Software verification with validation of results (Report on SV-COMP 2017). In *Proc. TACAS*, LNCS 10206, pages 331–349. Springer, 2017.
5. D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proc. ICSE*, pages 326–335. IEEE, 2004.
6. D. Beyer and M. Dangl. Verification-aided debugging: An interactive web-service for exploring error witnesses. In *Proc. CAV*, LNCS 9780. Springer, 2016.
7. D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proc. FSE*, pages 721–733. ACM, 2015.
8. D. Beyer, M. Dangl, and P. Wendler. Boosting k-induction with continuously-refined invariants. In *Proc. CAV*, LNCS 9206, pages 622–640. Springer, 2015.
9. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.
10. D. Beyer, A. Holzer, M. Tautschnig, and H. Veith. Information reuse for multi-goal reachability analyses. In *Proc. ESOP*, LNCS 7792, pages 472–491. Springer, 2013.
11. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
12. D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.
13. D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Proc. FASE*, LNCS 7793, pages 146–162. Springer, 2013.
14. D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: Requirements and solutions. *Int. J. Softw. Tools Technol. Transfer*, 2017.
15. D. Beyer and P. Wendler. Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In *Proc. SPIN*, LNCS. Springer, 2013.
16. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS*, LNCS 1579, pages 193–207. Springer, 1999.
17. M. Böhme, V. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *Proc. SIGSAC*, pages 1032–1043. ACM, 2016.
18. J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. ASE*, pages 443–446. IEEE, 2008.
19. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, pages 209–224. USENIX Association, 2008.
20. A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *Proc. TACAS*, LNCS 7795, pages 93–107. Springer, 2013.
21. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
22. E. M. Clarke, D. Kröning, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS*, LNCS 2988, pages 168–176. Springer, 2004.
23. W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957.
24. C. Csallner and Y. Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proc. ICSE*, pages 422–431. ACM, 2005.

25. M. Dangl, S. Löwe, and P. Wendler. CPAchecker with support for recursive programs and floating-point arithmetic. In *Proc. TACAS*, LNCS. Springer, 2015.
26. N. Eén and N. Sörensson. An extensible SAT solver. In *Proc. SAT*, LNCS 2919, pages 502–518. Springer, 2003.
27. M. Y. R. Gadelha, H. I. Ismail, and L. C. Cordeiro. Handling loops in bounded model checking of C programs via k-induction. *STTT*, 19(1):97–114, 2017.
28. S. J. Galler and B. K. Aichernig. Survey on test data generation tools. *STTT*, 16(6):727–751, 2014.
29. P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. PLDI*, pages 213–223. ACM, 2005.
30. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proc. NDSS*. The Internet Society, 2008.
31. S. Graf and H. Saïdi. Construction of abstract state graphs with Pvs. In *Proc. CAV*, LNCS 1254, pages 72–83. Springer, 1997.
32. M. Heizmann, D. Dietsch, J. Leike, B. Musa, and A. Podelski. Ultimate Automizer with array interpolation. In *Proc. TACAS*, LNCS 9035, pages 455–457. Springer, 2015.
33. M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In *Proc. CAV*, LNCS 8044, pages 36–52. Springer, 2013.
34. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
35. A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. How did you specify your test suite? In *Proc. ASE*, pages 407–416. ACM, 2010.
36. K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jFuzz: A concolic whitebox fuzzer for Java. In *Proc. NFM*, pages 121–125, 2009.
37. R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4), 2009.
38. A. V. Khoroshilov, V. Mutilin, A. K. Petrenko, and V. Zakharov. Establishing Linux driver verification process. In *Proc. Ershov Memorial Conference*, LNCS 5947, pages 165–176. Springer, 2009.
39. Y. Köroglu and A. Sen. Design of a modified concolic testing algorithm with smaller constraints. In *Proc. ISSTA*, pages 3–14. ACM, 2016.
40. D. Kröning and M. Tautschnig. Cbmc: C bounded model checker (competition contribution). In *Proc. TACAS*, LNCS 8413, pages 389–391. Springer, 2014.
41. K. Li, C. Reichenbach, C. Csallner, and Y. Smaragdakis. Residual investigation: Predictive and precise bug detection. In *Proc. ISSTA*, pages 298–308. ACM, 2012.
42. K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV*, LNCS 2725, pages 1–13. Springer, 2003.
43. J. Morse, M. Ramalho, L. Cordeiro, D. Nicole, and B. Fischer. Esbmc 1.22 (competition contribution). In *Proc. TACAS*, LNCS 8413. Springer, 2014.
44. Z. Pavlinovic, A. Lal, and R. Sharma. Inferring annotations for device drivers from verification histories. In *Proc. ASE*, pages 450–460. ACM, 2016.
45. K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for C. In *Proc. ESEC/FSE*, pages 263–272. ACM, 2005.
46. H. Seo and S. Kim. How we get there: A context-guided search strategy in concolic testing. In *Proc. FSE*, pages 413–424. ACM, 2014.
47. I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. Novikov, A. K. Petrenko, and A. V. Khoroshilov. Configurable toolset for static verification of operating systems kernel modules. *Programming and Computer Software*, 41(1):49–64, 2015.